Domain decomposition methods. Application to high-performance computing

Pierre Jolivet

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **mathématiques appliquées**

Arrêté ministériel : du 7 août 2006

Présentée par

Pierre Jolivet

Thèse dirigée par **Christophe Prud'homme** et **Frédéric Nataf**
et coencadrée par **Frédéric Hecht**

préparée au sein du **Laboratoire Jean Kuntzmann**
et du **Laboratoire Jacques-Louis Lions**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Méthodes de décomposition de domaine. Application au calcul haute performance.

Thèse soutenue publiquement le **2 octobre 2014**,
devant le jury composé de :

George Biros
Professeur The University of Texas at Austin, Rapporteur
Olaf Schenk
Professeur Università della Svizzera italiana, Rapporteur
Eric Blayo
Professeur Université Joseph Fourier, Examinateur
Victorita Dolean
Maître de conférences Université Nice Sophia Antipolis, Examinatrice
Luc Giraud
Directeur de recherche Inria, Examinateur
Christophe Prud'homme
Professeur Université de Strasbourg, Directeur de thèse
Frédéric Nataf
Directeur de recherche CNRS, Co-Directeur de thèse
Frédéric Hecht
Professeur Université Pierre et Marie Curie, Co-Encadrant

Remerciements

C'est avec une grande émotion que j'adresse mes premiers remerciements à mes directeurs de thèse, Frédéric et Christophe. Vous m'avez beaucoup appris pendant ces trois années, et je n'aurais pas pu progresser et m'épanouir autant dans mon travail de recherche sans votre soutien et nos échanges nombreux. Votre cohésion autour de mes travaux m'a permis d'avancer à mon rythme et dans les meilleures conditions. Je souhaite également remercier Frédéric Hecht, tant pour sa bonne humeur que pour sa disponibilité. Enfin, une partie non négligeable des résultats présentés dans ce manuscrit n'auraient pas pu voir le jour sans l'aide de Laura Grigori : merci de m'avoir lancé dans le monde du calcul haute performance.

Merci à Olaf Schenk et George Biros d'avoir accepté d'être les rapporteurs de ma thèse, ainsi qu'à Eric Blayo, Victorita Dolean et Luc Giraud de faire partie du jury. J'ai eu la chance de suivre vos recherches voire même de travailler avec certains d'entre vous, et je suis honoré que vous ayez accepté de participer à l'évaluation de cette thèse.

Pour les nombreuses discussions intéressantes, je témoigne ma reconnaissance aux membres du Laboratoire Jacques-Louis Lions, en particulier ceux du bureau 15-25-302 mais aussi Marie, Nicole, Ryadh et Francisco. Je pense aussi aux membres de l'équipe Inria Alpines : Long, Mathias, Sophie, Rémi et Sébastien, et enfin aux personnes qui travaillent autour de Feel++, dont Adboulaye et Vincent.

Je profite également de ce moment pour remercier toutes les équipes "support" que j'ai côtoyées, dans les laboratoires, notamment Khashayar, Philippe, Catherine, Nadine, Salima (LJLL), Hélène, Juana (LJK) et Laurence (Inria), ou dans les centres de calcul au CCRT[1], TGCC[2] et à l'IDRIS[3].

Je tiens aussi à exprimer ma gratitude à ceux qui m'ont facilité beaucoup de démarches logistiques lors de mes déplacements, en France ou à l'étranger. Parmi les personnes que je n'ai pas encore citées, merci à Michael, Nesrine, George et Thomas.

Rien de tout cela n'aurait été possible sans l'aide des familles Jolivet et Zitoun, et tout particulièrement celle de Danièle, Christian, Nathalie et Julien. Un grand merci à Aurélie pour son soutien inconditionnel ainsi qu'à mes amis de longue date qui ont toujours été présents quand il le fallait : Antoine, Clothilde et Thibault, Emmanuelle et Paul, Sonia et Clément, Benjamin, Julien, Amaury, Xavier, Jérôme…

Pour finir et par ce manuscrit, je rends hommage à mes grands-parents, Henriette, Geneviève et Dominique.

[1] Centre de Calcul Recherche et Technologie.
[2] Très Grand Centre de calcul du CEA.
[3] Institut du Développement et des Ressources en Informatique Scientifique.

Contents

Introduction **1**
i Version française . . . 1
i.1 Contexte . . . 1
i.2 Résumé et contributions . . . 3
ii English version . . . 6
ii.1 Context . . . 6
ii.2 Summary and contributions . . . 8

1 Domain decomposition preconditioners and HPC **13**
1.1 Overlapping Schwarz methods . . . 14
1.1.1 Towards an efficient formulation for distributed computing . . . 19
1.1.2 One-level methods . . . 21
1.1.3 Two-level methods . . . 22
1.2 Substructuring methods . . . 23
1.2.1 Balancing Neumann-Neumann preconditioner . . . 29
1.2.2 FETI preconditioner . . . 31
1.3 Improving preconditioners using spectral information: GenEO . . . 33
1.3.1 Overlapping methods . . . 34
1.3.2 Substructuring methods . . . 35

2 A unified framework **37**
2.1 Software design . . . 38
2.1.1 Parallel programming model . . . 38
2.1.2 Basic linear algebra . . . 38
2.1.3 Linear solvers . . . 40
2.1.4 Eigenvalue solvers . . . 41
2.1.5 Graph partitioners . . . 42
2.1.6 Similar libraries . . . 42
2.2 Necessary objects . . . 43
2.2.1 Subdomain . . . 43
2.2.2 Coarse operator . . . 46
2.2.3 Sparse and dense eigensolvers for GenEO . . . 50
2.2.4 Iterative methods . . . 51
2.3 A simple prototype . . . 54
2.3.1 Decomposition and partition of unity . . . 54
2.3.2 Finite difference matrices and deflation vectors . . . 56
2.3.3 Instantiation of the preconditioner . . . 57

3 Finite element languages **59**
3.1 FreeFem++ . . . 60
3.1.1 Mesh generation and decomposition . . . 61
3.1.2 Matrix assembly . . . 62
3.1.3 Transfer operators . . . 63
3.1.4 Interface with the framework . . . 65
3.1.5 Generating a global numbering . . . 66
3.1.6 Nonlinear and time-dependent solid mechanics . . . 66
3.2 Feel++ . . . 75
3.2.1 Preprocessing steps . . . 76
3.2.2 Transfer operators . . . 77
3.2.3 Retrieving a local matrix . . . 78
3.2.4 Calling the solver . . . 78

4 Improving scalability **81**
4.1 Distribution of the coarse operator . . . 81
4.2 Subdomain-level parallelism . . . 86
4.3 Separation of tasks . . . 86
4.4 Synchronization-avoiding Krylov solver . . . 87

5 Numerical experiments **91**
5.1 Test bed . . . 91
5.2 Comparison with multigrid solvers . . . 98
5.3 Comparison with direct solvers . . . 99

Conclusion **103**

Appendices **105**
A Gmsh geometries . . . 105
B Structural UML diagram of the library . . . 108
C Two published papers . . . 110

Bibliography **111**

List of Figures and Algorithms

1 Frequency and number of transistors of some Intel processors 6
2 Performance of the #1 on the TOP500 list and its number of processors . . 6
3 Wall-clock times spent in various steps of a complete physical simulation . 7

1.1 Original geometry used to introduce the alternating Schwarz method . . . 15
1.2 Decomposition of Ω into subdomains with different levels of overlap . . . 16
1.3 Initial numbering of the finite element space V and decomposition of Ω . . 18
1.4 Local numbering of each local finite element space 18
1.5 Substructuring of Ω into three subdomains 24
1.6 Trace operators used in substructuring methods 25
1.7 Primal assembly operators defined on the interface of each subdomain . . 26
1.8 Redundant jump operators used for imposing constraints 27

2.1 Local operations for applying the global operator in overlapping methods . 44
2.2 Communications for applying the global operator in overlapping methods 44
2.3 Local operations for applying the global BDD operator 44
2.4 Local operations for applying the global FETI operator 44
2.5 Communications for applying the global FETI operator 45
2.6 Different distributions of the coarse operator 49
2.7 Representation of the four operations performed during one coarse correction 50
2.8 Preconditioned Conjugate Gradient 52
2.9 Preconditioned Generalized Minimal RESidual method 53

3.1 A truncated and refined mesh generated by FreeFem++ 60
3.4 Examples of sixteen-way partitionings of a square domain 62
3.5 Construction of the local matrix and local right-hand side in FreeFem++ . 63
3.6 Construction of the overlapping transfer operators in FreeFem++ 64
3.7 Construction of the nonoverlapping transfer operators in FreeFem++ . . . 65
3.8 Nonlinear elasticity kinematic operators computed with FreeFem++ 73
3.9 Initialization of the local finite element space with Feel++ 76
3.10 Numbering of the interfaces with the neighboring subdomains with Feel++ 77
3.11 Renumbering of the transfer operators with Feel++ 77
3.12 Assembling and retrieving the local matrix and right-hand side with Feel++ 78
3.14 Building the local rigid body modes with Feel++ 79

4.1 Schematic construction of the coarse operator 84
4.2 Schematic assembly of the coarse operator on master processes 84
4.3 Sixteen subdomains split among four masters 85
4.4 Time diagram showing various tasks performed by a two-level method . . 87
4.5 Computational loop of the p^1-GMRES 88

4.6 Two possible workflows for the p^1-GMRES 89

5.1 Variations of the coefficients used for some three-dimensional experiments 92
5.2 Strong scaling experiments 94
5.3 Convergence of the GMRES for a problem of elasticity using 1 024 subdomains 94
5.4 Diffusivity used for the two-dimensional weak scaling experiment 95
5.5 Weak scaling experiments 97
5.6 Timings for assembling and factorizing coarse operators 97
5.7 Comparison of iterative solvers for solving Poisson's equation 98
5.8 Comparison of iterative solvers for solving the system of linear elasticity . 99
5.9 Comparison with direct solvers for solving Poisson's equation 100

A.1 A two-dimensional cantilever 105
A.2 A three-dimensional tripod 107
B.1 Structural UML diagram of the library 108

Introduction

This chapter gives a broad introduction to the field of sparse linear algebra. The need for efficient parallel solvers is motivated by concrete examples. The outline of this thesis as well as a list of its main contributions in the fields of domain decomposition methods and high-performance computing are available in section ii.2.

Ce chapitre est une est une brève introduction du domaine de l'algèbre linéaire creuse. On motive le besoin de solveurs parallèles efficaces par des exemples concrets. Le plan du manuscrit ainsi qu'une liste des contributions de cette thèse dans les méthodes de décomposition de domaine et du calcul haute performance sont disponibles section i.2.

Contents

i Version française 1

i.1 Contexte 1

i.2 Résumé et contributions 3

ii English version 6

ii.1 Context 6

ii.2 Summary and contributions 8

i Version française

i.1 Contexte

Le domaine du calcul scientifique évolue de manière très rapide depuis les années 1980 grâce à l'aide de projets innovants comme Netlib[4], MPI [Snir et al. 1995], PETSc [Balay, Gropp, et al. 1997], et d'autres. L'ensemble de ces recherches a permis aux mathématiciens appliqués de créer des outils puissants pour simuler des phénomènes physiques complexes comme dans [Alimi et al. 2012], en utilisant—si besoin est—un haut niveau d'abstraction. Par le passé, les performances de ces outils étaient principalement déterminées par la fréquence des composants sur lesquels les simulations étaient exécutées. Malheureusement, depuis les années 2000, la fréquence des unités de calcul n'a pas crû de manière aussi soutenue que par le passé à cause d'une trop grosse consommation énergétique, cf. figure 1. Le calcul parallèle est ainsi devenu un paradigme important en architecture informatique, comme indiqué figure 2. Celui-ci est déjà utilisé de manière récurrente en calcul haute performance

[4] URL : http ://www.netlib.org/.

sur les supercalculateurs. De plus, à cause du ralentissement de la croissance des fréquences, son utilisation devient également nécessaire sur des architectures moins évoluées, comme les ordinateurs de bureau.

Un effet direct de ce changement est qu'il peut devenir plus délicat de fournir un haut niveau d'abstraction ou d'expressivité dans les logiciels numériques parallèles. Pour cette raison, les utilisateurs se tournent vers des algoritmes boîtes noires pour les routines les plus gourmandes en temps de calcul, par exemple le calcul de la solution d'un système linéaire dans une méthode de discrétisation implicite comme celle des éléments finis, cf. figure 3.

Historiquement, il existe deux classes de méthodes pour résoudre des systèmes linéaires : les méthodes directes décrites dans [Duff, Erisman et Reid 1986] et les méthodes itératives détaillées dans [Saad 2003]. D'une part, les méthodes directes sont connues pour être capables de résoudre tout système inversible en un nombre fini d'opérations si l'on suppose l'absence d'erreurs d'arrondis. Ainsi, elles sont les solveurs les plus robustes, mais la quantité de mémoire qu'elles nécessitent pour trouver de telles solutions peut se montrer excessive dans le cas de problèmes très grands ou difficiles. D'autre part, les méthodes itératives consomment très peu de mémoire puisqu'elles n'ont besoin de stocker que quelques vecteurs de la taille du problème à résoudre. En revanche, elles sont moins robustes et peuvent même ne jamais converger vers une solution adéquate en fonction du système. C'est pour cette raison qu'il est primordial de préconditionner les systèmes résolus par méthodes itératives. Ce procédé consiste à modifier les systèmes sus-cités sous une forme qui rend leur résolution plus aisée. Mathématiquement parlant, si l'on cherche à résoudre un système algébrique $Ax = b$ en utilisant une méthode itérative, il s'avère souvent être plus efficace de trouver un préconditionneur M^{-1} approchant A^{-1} et de résoudre $M^{-1}Ax = M^{-1}b$. Le conditionnement de $M^{-1}A$ est alors bien plus faible que celui de A. Développer ou choisir le bon préconditionneur pour un problème donné est une tâche délicate mais qui peut diminuer de manière conséquente le temps d'exécution des logiciels numériques. Rendre la construction d'un préconditioneur indépendant du problème sous-jacent tout en gardant de bonnes propriétés numériques est d'autant plus délicat. On qualifie en général de tels préconditionneurs de purement algébriques, dont voici une liste succinte de certains des plus connus : sparse approximate inverse [Grote et Huckle 1997], factorisation incomplète [Saad 1994]...

Récemment, des préconditionneurs quasi-optimaux ont été introduits dans le domaine des méthodes multigrilles [Brandt 1977] et dans celui des méthodes de décomposition de domaine [Giraud et Haidar 2009 ; Smith, Bjørstad et Gropp 2004]. Ces deux classes d'algorithmes sont généralement décrites comme hybrides. Cela s'explique par le fait que leur but est de préconditionner un système pour une méthode itérative, tout en utilisant des méthodes directes dans des "sous-systèmes" ou des problèmes auxiliaires pour la définition du préconditionneur global. Une telle hybridation permet typiquement de s'assurer que ces méthodes réspectent trois points importants pour les algorithmes parallèles.

1. Le temps nécessaire pour effectuer des opérations concurrentes par des unités de calcul différentes tend à être largement supérieur à celui passé à échanger des informations entre processus ou à synchroniser. On parle de ratio calcul-sur-communication.

2. La quantité de mémoire additionnelle induite par la parallèlisation des méthodes, e.g. pour créer des buffers pour les transferts entre processus, est plutôt faible, surtout comparée à des méthodes directes parallèles. Cela permet aussi de garantir une meilleure localité de la mémoire, ce qui peut réduire l'effet du memory wall, cf. [Wulf et McKee 1995].

3. En fonction de la distribution du problème initial, ces algorithmes ont naturellement une charge équilibrée, ainsi, les unités de calcul ne sont pas sujettes à une surcharge et peuvent optimiser l'utilisation des ressources, maximiser leur rendement, et minimiser le temps de réponse.

Il en résulte des méthodes fortement parallèles qui sont assez robustes pour résoudre des problèmes complexes, comme par exemple dans [Biros et Ghattas 2005 ; Klawonn et Rheinbach 2010 ; Sundar et al. 2012]. Dans cette thèse, des méthodes de décomposition de domaine pour des types de discrétisation conformes tels que la méthode des différences finies sur grilles régulières ou la méthode des éléments finis sur maillages conformes seront étudiées. Elles sont utilisées pour résoudre un problème défini sur un domaine global en le découpant en plusieurs sous-problèmes de plus petites tailles et en itérant pour uniformiser la solution entre des sous-domaines voisins. Cette approche est basée sur le paradigme "diviser pour régner". Puisque tous les sous-problèmes sont indépendants et que les sous-domaines communiquent uniquement à travers leur interface ou une petite zone de recouvrement pour calculer la solution globale, les préconditionneurs par décomposition de domaine ont ainsi un ratio calcul-sur-communication élevé, cf. item 1. L'utilisation d'un partitionneur automatique de graphes permet la vérification de critères. Cette dernière rend la décomposition du domaine global équilibrée, cf. item 3, en des sous-domaines ayant par exemple le même nombre de points de grille ou d'éléments de maillage. Malgré tous ces points forts, les méthodes de décomposition de domaine sont rarement disponibles de façon autonome, notamment pour les raisons suivantes :

- les méthodes les plus avancées sont rarement purement algébriques et sont liées à la méthode de discrétisation sous-jacente nécessaire pour l'assemblage du problème,
- il n'est pas toujours facile de détecter l'interface entre sous-domaines, ni d'avoir une structure logicielle cohérente pour l'échange de donnés entre ceux-ci.

i.2 Résumé et contributions

Cette thèse contribue aux méthodes de décomposition de domaine et au calcul haute performance comme détaillé ci-dessous.

Dans le chapitre 1, il est proposé au lecteur une introduction aux préconditionneurs par décomposition de domaine. En utilisant une approche standard, on montre que leur formalisme n'est pas nécessairement bien adapté au calcul distribué. La nouvelle formulation proposée permet d'accroître le niveau de parallèlisme tant pour la construction des préconditionneurs que pour certaines étapes de prétraitement comme la génération de maillage. Il est par exemple possible, grâce à ce formalisme, d'utiliser un mailleur et un noyau éléments finis, tous deux séquentiels, tout en effectuant des simulations parallèles à grandes échelles, tant qu'un raffinement adaptatif de maillage ou de grille n'est pas nécessaire. Cette vision unifiée des méthodes de décomposition de domaine est particulièrement adaptée pour deux familles majeures de préconditionneurs : les méthodes de Schwarz avec recouvrement et les méthodes par sous-structuration. Dans leurs versions les plus basiques, ces préconditionneurs peuvent rencontrer des difficultés pour la résolution de problèmes industriels complexes car ils ne sont pas assez robustes. Une approche pour pallier à ce problème est également présentée dans ce premier chapitre. Elle se base sur l'utilisation d'opérateurs de projection qui sont engendrés par des vecteurs propres calculés parallèlement. On les obtient *a priori* en résolvant des problèmes aux valeurs propres généralisées. Il s'agit

de la méthode GenEO, pour Generalized Eigenvalue problem on the Overlap, et il a été prouvé dans des travaux antérieurs [Spillane, Dolean, et al. 2013 ; Spillane et Rixen 2013] qu'elle est théoriquement stable.

Dans le chapitre 2, une librairie unifiée pour utiliser les méthodes de décomposition de domaine comme des solveurs en boîte noire est présentée. On se focalisera principalement sur sa généricité et son abstraction qui permettent :

1. aux utilisateurs d'essayer une méthode ou une autre sans difficulté,
2. aux développeurs de maintenir et d'optimiser plus facilement la librairie,
3. de comparer sur de mêmes cas tests une grande gamme de méthodes,
4. de ne pas être lié à la méthode de discrétisation sous-jacentes pour pouvoir générer des préconditionneurs en utilisant des entrées algébriques.

Les ingrédients nécessaires pour utiliser les méthodes de décomposition de domaine sont rappelés dans un premier temps. La librairie est ensuite expliquée en détail : implémentée en C++, elle utilise MPI pour le passage de messages ainsi que OpenMP pour le parallèlisme au niveau des sous-domaines. L'implémentation est en accord avec le formalisme algébrique introduit dans le premier chapitre. À la fin du chapitre, un prototype autonome est expliqué. Écrit également en C++, il peut résoudre avec une méthode de Schwarz avec recouvrement un simple problème aux limites défini sur une grille rectangulaire en utilisant la méthode des différences finies. Cela permet d'expliciter les différentes étapes à effectuer pour instancier la libraire.

Dans le chapitre 3, la librairie est interfacée avec deux codes éléments finis bien établis :

- FreeFem++, un langage dédié qui permet la discrétisation de formulations variationnelles d'équations aux dérivées partielles avec la méthode des éléments finis en utilisant son propre langage interprété,
- Feel++, un langage dédié embarqué dans le C++ qui peut être appelé à partir d'un code Python ou C++ pour la résolution d'équations aux dérivées partielles avec des méthodes de Galerkin généralisées.

Encore une fois, les avantages de la reformulation effectuée dans le chapitre 1 et la généricité de la libraire du chapitre 2 sont bien visibles car les deux codes éléments finis varient sur plusieurs points. Les méthodes de décomposition de domaine peuvent être appelées facilement :

- depuis FreeFem++, en rajoutant des mots-clefs dans le langage pour manipuler les préconditionneurs sans effort,
- dans Feel++, en utilisant judicieusement ses structures de données internes sophistiquées et en inférant ainsi les informations nécessaires pour initialiser et utiliser les préconditionneurs.

Dans le chapitre 4, des stratégies avancées pour améliorer le passage à l'échelle de la librairie introduite dans le chapitre 2 sont présentées. Un nouvel algorithme pour assembler et pour utiliser les opérateurs de projection définis dans le chapitre 1 est d'abord étudié. Cela engendre une méthode optimale en terme de nombre de messages et d'échanges entre processus. Ensuite, on montre comment les méthodes de décomposition de domaine peuvent bénéficier d'un parallèlisme à granularité plus

fine en utilisant le multithreading pour accélérer des calculs locaux à chaque sous-domaine. On explique également comment utiliser les opérateurs de projection de façon asynchrone. C'est un point important pour minimiser la surcharge des processus responsables de ces opérateurs, entrainant ainsi une meilleure répartition de charge. Pour finir, ces améliorations peuvent être combinées pour fournir un solveur itératif avec peu de synchronisation. Cette classe de méthode devient cruciale pour le passage à l'échelle sur les architectures extrêmes où le coût des communications et des synchronisations croient bien plus rapidement que celui du coût arithmétique de ces méthodes.

Dans le chapitre 5, les performances du code développé lors de cette thèse sont évaluées et l'on observe des résultats quasi-optimaux jusqu'à 16 384 threads pour résoudre des problèmes elliptiques bi- et tridimensionnels exécutés sur Curie, un supercalculateur français PFLOP/s-ique. Un test de passage à l'echelle fort (resp. faible) illustre comment la librairie se comporte lorsque le nombre de processus pour résoudre des problèmes globaux (resp. des problèmes locaux) de taille fixe augmente. Ce chapitre se termine par deux comparaisons avec des solveurs de pointe pour certifier le potentiel de la librairie :

- d'une part, les méthodes de Schwarz avec recouvrement étudiées dans le chapitre 1 et des préconditionneurs multigrilles sont utilisés pour résoudre des systèmes linéaires distribués sur 4 192 processus,
- d'autre part, les préconditionneurs par sous-structuration et des solveurs directs sont comparés pour la résolution sur 2 à 64 processus d'un problème global fixe de cinq millions d'inconnues.

Tous ces résultats numériques prouvent l'efficacité de la méthodologie détaillée dans les chapitres précédents pour résoudre des problèmes de quelques millions à plusieurs milliards d'inconnues.

En conclusion, des directions pour des recherches futures sont proposées.

Le reste de ce manuscrit est rédigé en anglais, à l'exception des chapeaux de chaque chapitre qui sont également traduits.

ii English version

ii.1 Context

The field of scientific computing is evolving at an impressive rate since the 1980s thanks to the help of pioneering projects like Netlib[5], MPI [Snir et al. 1995], PETSc [Balay, Gropp, et al. 1997], and such. Altogether, these pieces of research help applied mathematicians create powerful tools for performing advanced numerical simulations as in [Alimi et al. 2012], using a—possibly—high-level of abstraction. In the past, their throughputs were mainly determined by the frequencies of the chips on which the simulations were running. Unfortunately, since the mid-2000s, frequencies of computing units have not scaled as steadily as before due to an increased power consumption, see fig. 1.

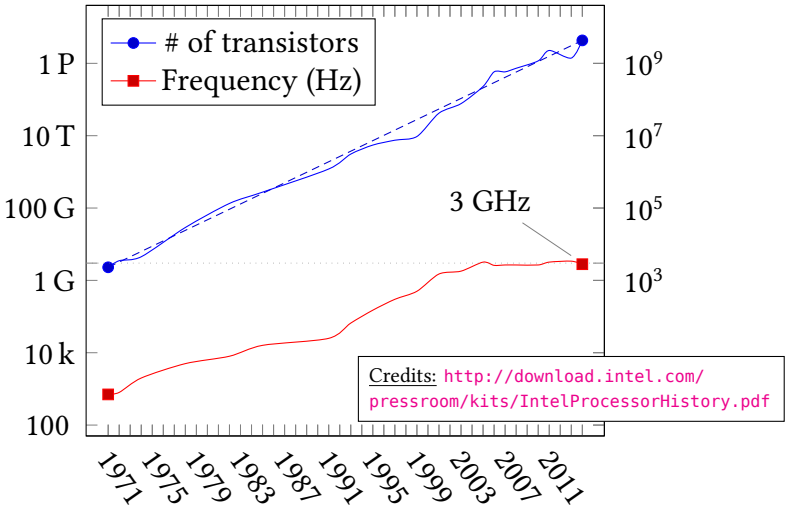


Fig 1: Frequency and number of transistors of some Intel processors clearly showing the end of frequency scaling around the 3 GHz mark.

Parallel computing has now become the dominant paradigm in computer architecture, as for example displayed in fig. 2. While it has been employed for many years in the field of high-performance computing on supercomputers, it is also becoming a necessity on less sophisticated architectures like desktop computers because of the end of frequency scaling.

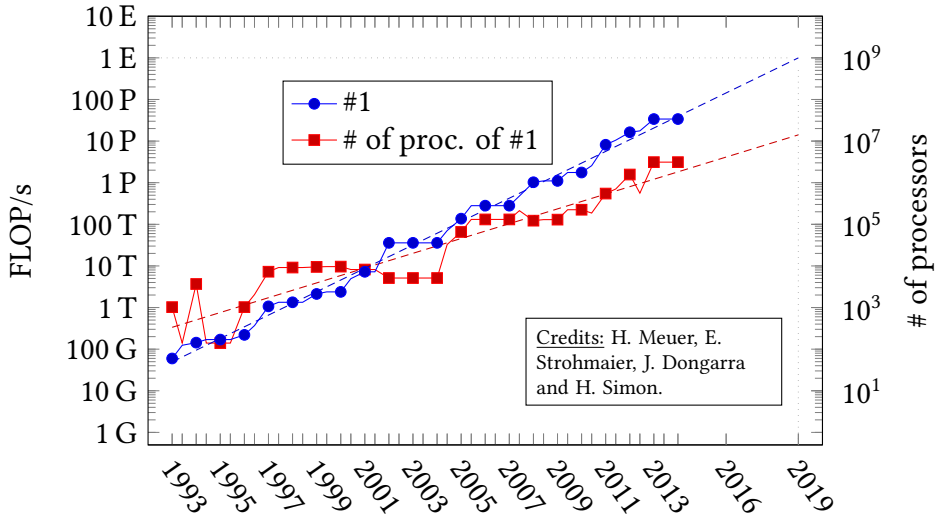


Fig 2: Performance of the #1 on the TOP500 list over time, and its number of processors. The list is used to benchmark supercomputers.

[5] URL: http://www.netlib.org/.

A direct effect of this switch is the fact that it can now be harder to provide a high-level of abstraction or expressivity with parallel numerical software. For that reason, end-users are favoring black box algorithms for most time-consuming routines, such as computing the solution of a linear system in an implicit discretization method like the finite element method, cf. fig. 3.

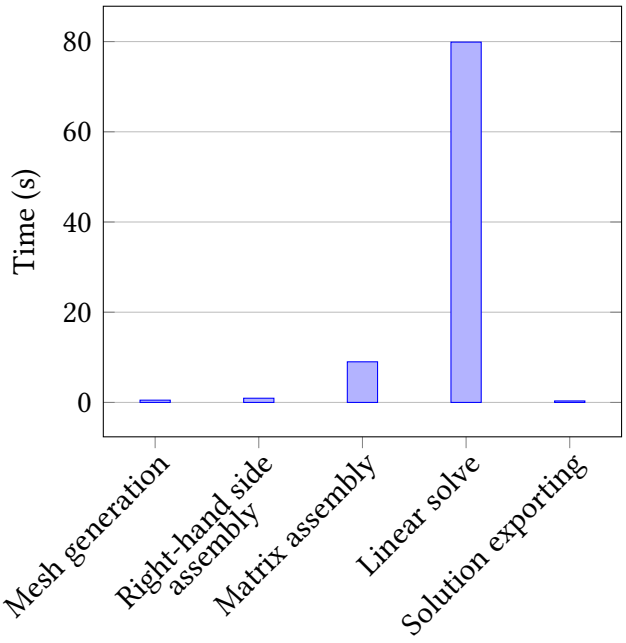


Fig 3:
Wall-clock times spent in various steps of a complete finite element simulation using Feel++ [Prud'homme 2006] for solving Stokes equations in three dimensions.

There are historically two classes of methods for solving linear systems: direct methods described in [Duff, Erisman, and Reid 1986] and iterative methods depicted in [Saad 2003]. On the one hand, direct methods are known to be able to find a solution to any nonsingular problem in a finite number of operations, not taking round-off errors into account. In that sense, they are the most robust solvers, but the memory requirements to compute such a solution can sometimes be excessive when looking at really large or complex problems. On the other hand, iterative methods have really low memory consumption since they usually only need to store few problem-sized vectors, but they are less robust and might not always converge to an appropriate solution depending on the problem. Therefore, it is of paramount importance to precondition a system when using iterative methods, that is, to transform the said system into a form that is more suitable for solution. Mathematically speaking, if one tries to solve the algebraic system of equations $Ax = b$ using an iterative method, then it is commonly far more efficient to find a suitable preconditioner M^{-1} that approximates A^{-1} and solve $M^{-1}Ax = M^{-1}b$, where the condition number of $M^{-1}A$ is much lower than the one of A. Developing or choosing the right preconditioner for a given problem is a hard task in itself but can lead to tremendous improvements in numerical software runtimes. Making the construction of a preconditioner oblivious to the problem underlying the system while maintaining good numerical properties is even harder. Such oblivious preconditioners are often referred to as fully algebraic, and here are some of the most established: sparse approximate inverse [Grote and Huckle 1997], incomplete factorization [Saad 1994]...

Recently, near-optimal preconditioners have been introduced in the field of multigrid methods [Brandt 1977] and domain decomposition methods [Giraud and Haidar 2009; Smith, Bjørstad, and Gropp 2004]. These two groups of algorithms are often described as hybrid methods. That is because they are ultimately used to precondition a system for an iterative method, but direct methods are also employed within the definition of the global preconditioner on some smaller "subsystems" or auxiliary problems. Such hybridization usually ensures that these preconditioners comply with three important features of parallel algorithms.

1. The time spent carrying out concurrent operations by different processing units tends to be much higher than the one spent exchanging inter-process information or synchronizing. This is referred to as the computation-to-communication ratio.

2. The memory overhead induced by the parallelization of these methods, e.g. for buffering inter-process transfers, is rather low, especially when comparing with parallel direct methods. They can also guarantee better locality of reference, which may reduce the effect of the memory wall, cf. [Wulf and McKee 1995].

3. Depending on the distribution of the initial problem, these algorithms are naturally load balanced, which means that each processing unit is not subject to overload and can optimize resource use, maximize throughput, and minimize response time.

This results in highly concurrent methods that are robust enough to solve complex problems, see [Biros and Ghattas 2005; Klawonn and Rheinbach 2010; Sundar et al. 2012] for examples. In this thesis, domain decomposition methods for conforming discretization techniques such as the finite difference method on a regular grid or the finite element method on conforming meshes will be studied. They are used to solve a problem defined on a global domain by splitting it into smaller problems on subdomains and iterating to coordinate the solution between neighboring subdomains: this is based on the "divide & conquer" paradigm. Because all smaller problems are independent and that neighboring subdomains only communicate through their interfaces or a small overlapping region for computing the global solution, domain decomposition preconditioners indeed have a high computation-to-communication ratio, cf. item 1. Using automatic graph partitioners, it is possible to verify some criteria that will result in a balanced split of the global domain, cf. item 3, into subdomains that have for example the same number of grid points or mesh elements. Despite all these favorable factors, domain decomposition methods are seldom available as standalone solvers for few reasons:

- most advanced methods are not purely algebraic and are linked to the underlying discretization technique used to assemble the problem,

- it is not always easy to detect the interface between subdomains and have a coherent software architecture to exchange data between these.

ii.2 Summary and contributions

This thesis makes contributions to the fields of domain decomposition methods and high-performance computing as detailed next.

In chapter 1, the reader is introduced to domain decomposition preconditioners. Using a standard approach, it is shown that the formalism does not necessarily fit with distributed computing. A new formulation is proposed to provide an increased concurrency, both for the setup of the preconditioners, but also for the preprocessing steps such as mesh generation. As an example, it is possible, thanks to this formulation, to use a sequential mesher and sequential finite element kernel and still be able to perform large-scale parallel experiments, as long as there is no need for adaptive grid or mesh refinement. This unified vision of domain decomposition methods is particularly well-suited for two major families of preconditioners: overlapping Schwarz methods and substructuring methods. In their most basic versions, these preconditioners may face difficulties for solving challenging industrial problems because they are not robust enough. An approach for addressing this hurdle is also described in the first chapter. It is based on the use of projection operators which are spanned by eigenvectors computed concurrently. They are obtained a priori by solving local generalized eigenvalue problems. This method is called GenEO, for Generalized

Eigenvalue problem on the Overlap, and was proven to be theoretically scalable in other prior works [Spillane, Dolean, et al. 2013; Spillane and Rixen 2013].

In chapter 2, a unified framework for using these domain decomposition methods as black box solvers is presented. Emphasis is put into genericity and abstraction, so that:

1. it is convenient for a user to try one method or another one,
2. it easier to maintain and optimize the framework as a developer,
3. comparing a wide range of methods on the same problem is achievable without great effort,
4. there is no link with the underlying discretization technique so that the preconditioners can be created using algebraic inputs.

The necessary ingredients for performing domain decomposition methods are recalled first. Then, the framework is described in great detail: implemented in C++, it utilizes MPI for message passing as well as OpenMP for subdomain-level parallelism. The implementation is congruous with the algebraic formalism introduced in the first chapter. At the end of the chapter, a standalone prototype is explained. Written also in C++, it can solve with an overlapping Schwarz method a simple boundary value problem on a rectangular grid using the finite difference method. It is helpful to present the different steps that need to be conducted to instantiate the framework.

In chapter 3, the framework is interfaced with two long-established finite element libraries:

- FreeFem++, a domain-specific language that lets users discretize variational formulations of partial differential equations using the finite element method using its own interpreted language,
- Feel++, a domain-specific embedded language inside C++ that may be called from Python or directly from C++ for solving partial differential equations using generalized Galerkin methods.

Once again, the benefits of the reformulation done in chapter 1 and of the genericity of the framework in chapter 2 are clearly displayed since both libraries differ in various aspects. The domain decomposition methods can be easily called:

- in FreeFem++, by adding new keywords in the language to manipulate the preconditioners effortlessly,
- in Feel++, by taking advantage of the sophisticated internal data structures of the library and inferring the necessary information for initializing and using the preconditioners.

In chapter 4, advanced strategies for further improving the scalability of the framework introduced in chapter 2 are presented. A novel algorithm for assembling and using the projection operators defined in chapter 1 is studied first. It leads to an optimal method in terms of number of messages and inter-process exchanges. Then, it is shown how domain decomposition method can benefit from fine-grained parallelism using multithreading for accelerating computations local to each subdomain. It is furthermore explained how it is possible to use the projection operators asynchronously. This is important for minimizing the overload of the processes in charge of these operators, and in the end, induce better load-balancing. Eventually, these enhancements may

be combined to provide a synchronization-avoiding iterative solver. This class of algorithms is becoming crucial for scaling on extreme-scale architectures where the communication and synchronization costs are growing at a much faster pace than the arithmetic cost of these methods.

In chapter 5, the performance of the code developed during this thesis are evaluated and almost optimal scaling results are observed on up to 16 384 threads for solving bi- and tridimensional elliptic problems executed on Curie, a French PFLOP/s supercomputer. A strong (resp. weak) scaling experiment illustrates how the framework behaves with increasing number of processes and threads for fixed global problem sizes (resp. fixed problem sizes per processes and threads). Eventually, two comparisons with cutting-edge solvers assess the potential of the framework:

- on the one hand, overlapping Schwarz methods studied in chapter 1 and multigrid preconditioners solve linear systems on 4 192 processes,
- on the other hand, substructuring preconditioners and direct solvers are benchmarked for solving a fixed global problem of five million unknowns on 2 up to 64 processes.

All these numerical results prove the effectiveness of the workflow detailed in the preceding chapters for solving problems of few millions up to billions of unknowns.

As a conclusion, some directions for future research are presented.

This thesis led to the following publications, presentations, and awards.

Book

Dolean, V., P. Jolivet, and F. Nataf (2014). *An introduction to domain decomposition methods: algorithms, theory and parallel implementation*. SIAM, in preparation (cit. on pp. 13, 23, 35).

Journal papers

Dolean, V., P. Jolivet, F. Nataf, N. Spillane, and H. Xiang (2014). "Two-Level Domain Decomposition Methods for Highly Heterogeneous Darcy Equations. Connections with Multiscale Methods". In: *Oil Gas Sci. Technol. – Rev. IFP Energies nouvelles* 69.4, pp. 731–752.

Jolivet, P., V. Dolean, F. Hecht, F. Nataf, C. Prud'homme, and N. Spillane (2012). "High-performance domain decomposition methods on massively parallel architectures with FreeFem++". In: *Journal of Numerical Mathematics* 20.4, pp. 287–302 (cit. on pp. 23, 34, 59, 110).

Jolivet, P., F. Hecht, F. Nataf, and C. Prud'homme (2014b). "Scalable domain decomposition preconditioners for heterogeneous elliptic problems". In: *Scientific Programming* 22.2, pp. 157–171 (cit. on pp. 82, 110).

Conference papers

Jolivet, P., F. Hecht, F. Nataf, and C. Prud'homme (2013). "Scalable Domain Decomposition Preconditioners For Heterogeneous Elliptic Problems". In: *Proceedings of the 2013 ACM/IEEE conference on Supercomputing*. SC13. Best paper finalist. ACM, 80:1–80:11 (cit. on pp. 81, 82, 91, 110).

— (2014a). "Overlapping Domain Decomposition Methods with FreeFem++". In: *Domain Decomposition Methods in Science and Engineering XXI*. Vol. 98. Lecture Notes in Computational Science and Engineering. Springer, pp. 315–322 (cit. on p. 46).

Other conference presentations (for which no paper were submitted for publication in proceedings)

Jolivet, P. and F. Hecht (2013). "How to solve PDE easily with FreeFem++ ?" The European Numerical Mathematics and Advanced Applications (ENUMATH 2013).

Jolivet, P., F. Nataf, and C. Prud'homme (2013). "A unified framework for Schwarz and Schur methods". 22nd International Conference on Domain Decomposition Methods (DD22).

— (2014a). "Deflation-based domain decomposition preconditioners". SIAM Conference on Parallel Processing for Scientific Computing (PP14).

— (2014b). "Deflation-based domain decomposition preconditioners". 8th International Workshop on Parallel Matrix Algorithms and Applications (PMAA14).

— (2014c). "Deflation-based domain decomposition preconditioners". 11th World Congress on Computational Mechanics — 5th European Conference on Computational Mechanics — 6th European Conference on Computational Fluid Dynamics.

Grants and awards

Merit-based scholarship from Fondation Sciences Mathématiques de Paris to do three months of research abroad (4,500€). URL: http://www.sciencesmaths-paris.fr/en/.

SIAM Student Travel Award to attend the 2014 SIAM Conference on Parallel Processing for Scientific Computing ($800). See also [Jolivet, Nataf, and Prud'homme 2014a]. URL: http://www.siam.org/prizes/sponsored/travel.php.

Chapter 1

Domain decomposition preconditioners and high-performance computing

SOME fundamental aspects of domain decomposition methods are recalled in sections 1.1 and 1.2. Particular emphasis is placed on the reformulation of these methods, as partially done in [Dolean, Jolivet, and Nataf 2014], so that they can be used inside a high-performance framework. Section 1.3 gathers ways to improve standard preconditioners using generalized eigenvalue problems.

QUELQUES points clefs des méthodes de décomposition de domaine sont rappelés dans les sections 1.1 et 1.2. On insiste particulièrement sur la reformulation de ces méthodes, comme déjà partiellement effectuée dans [Dolean, Jolivet et Nataf 2014], de sorte qu'elles peuvent être utilisées dans une libraire à haute performance. La section 1.3 explique comment améliorer des préconditionneurs basiques en utilisant des problèmes aux valeurs propres généralisées.

Contents

1.1 Overlapping Schwarz methods . . . 14
1.1.1 Towards an efficient formulation for distributed computing . . . 19
1.1.2 One-level methods . . . 21
1.1.3 Two-level methods . . . 22
1.2 Substructuring methods . . . 23
1.2.1 Balancing Neumann-Neumann preconditioner . . . 29
1.2.2 FETI preconditioner . . . 31
1.3 Improving preconditioners using spectral information: GenEO . . . 33
1.3.1 Overlapping methods . . . 34
1.3.2 Substructuring methods . . . 35

It is presumed now that the Partial Differential Equation (PDE) being solved on a domain $\Omega \subset \mathbb{R}^d$ ($d = 2$ or 3) can be written in variational formulation, that is one needs to:

$$\text{find } u \in H_0^1(\Omega) \text{ such that } a(u, v) = l(v)\,, \tag{1.1}$$

where a is a bilinear symmetric coercive form and $l \in H_0^1(\Omega)^\star$ lies in the dual space.

Let $\mathcal{T} = \bigcup_{i=1}^{e} \{\mathcal{K}_i\}$ be a mesh of Ω, i.e. a tessellation of Ω made of e triangles, tetrahedra,

quadrilaterals... on which a finite set of n basis functions $\{\phi_i\}_{i=1}^n$ spans the finite element space V so that all functions of $u \in H^1(\Omega)$ can be discretized as:

$$u \approx u_h = \sum_{i=1}^{n} u_h[i]\phi_i \,, \tag{1.2}$$

where $\forall i \in [\![1; n]\!]$, $u_h[i]$ represents the value of the ith degree of freedom associated with the finite element function u_h. The finite element function u_h can be seen as a vector of scalar in $\mathbb{R}^n$ whose ith entry equals $u_h[i]$, so that throughout this document, the following abuse of notation will be made:

$$u_h \equiv \sum_{i=1}^{n} u_i \varepsilon_i \in \mathbb{R}^n \,,$$

where $\{\varepsilon_i\}_{i=1}^n$ is the canonical basis of $\mathbb{R}^n$. From now on, it will be accepted that all discretized functions can be seen as vectors, so that the subscript $_h$ will be dropped. The basis functions are frequently chosen as continuous linear functions ($\mathbb{P}_1$ finite elements) or continuous quadratic functions ($\mathbb{P}_2$ finite elements). The finite element method then leads to the following linear system:

$$Au = f, \tag{1.3}$$

where $(A_{ij})_{1 \leqslant i,j \leqslant n} = \int_\Omega a(\phi_j, \phi_i)$ and $(f_i)_{i=1}^n = \int_\Omega l(\phi_i)$.

It is now assumed that all the elements $\bigcup_{i=1}^{e} \{\mathcal{K}_i\}$ have been partitioned into N disjoint sets $\{\mathcal{T}_i\}_{i=1}^N$. Let Ω_i be the subdomain associated with $\mathcal{T}_i$, i.e. $\Omega_i = \bigcup_{\tau \in \mathcal{T}_i} \tau$, so that:

$$\Omega = \bigcup_{i=1}^{N} \Omega_i,$$
$$\forall i \in [\![1; e]\!], \exists! j \in [\![1; N]\!] \,:\, \mathcal{K}_i \in \mathcal{T}_j.$$

Let $\mathcal{O}_i$ be the set of neighboring subdomains, that is

$$\mathcal{O}_i = \{j \in [\![1; N]\!] \,:\, \text{at least one element of } \mathcal{T}_j \text{ touches } \mathcal{T}_i\}, \tag{1.4}$$

and additionally, let $\overline{\mathcal{O}}_i = \mathcal{O}_i \cup \{i\}$.

The notion of set will be used to represent an abstract data structure that stores unique values. Hence, for a set $\mathcal{S}$ of $\#\mathcal{S}$ integers[1], the following abuses of notation will be made:

- $\mathcal{S}(i)$ will refer to the ith element of the set, $\forall i \in [\![1; \#\mathcal{S}]\!]$,
- $\mathcal{S}^{-1}(s)$ will refer to the index of element whose value equals s, $\forall s \in \mathcal{S}$.

1.1 Overlapping Schwarz methods

Schwarz [1870] was interested in proving the existence of solutions to Poisson's equation given a source term f and a complex domain Ω:

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega \\ u &= 0 && \text{on } \partial\Omega \,. \end{aligned} \tag{1.5}$$

[1] $\#\mathcal{S}$ equals to the number of elements in the set $\mathcal{S}$.

The domain was at that time the union of two subdomains, a rectangle Ω_1 and a circle Ω_2 displayed fig. 1.1, so that the Fourier [1822] transform could be used independently to solve the equation on each regular geometry.

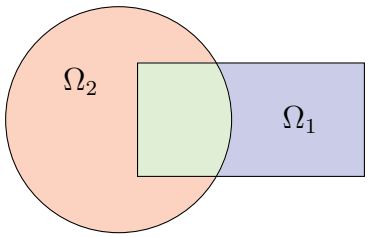


Fig 1.1: Original geometry used to introduce the alternating Schwarz method.

The problem was then to come up with a way to solve the equation on the complete domain Ω. This can be done by using an iterative method that exchanges information between subdomains at each step. This is called the alternating Schwarz method. Given a first iterate u_2^0, solve:

$$\begin{aligned} -\Delta u_1^{m+1} &= f && \text{in } \Omega_1 \\ u_1^{m+1} &= 0 && \text{on } \partial\Omega_1 \cap \partial\Omega \\ u_1^{m+1} &= u_2^m && \text{on } \Omega \setminus \Omega_1 \end{aligned} \qquad \begin{aligned} -\Delta u_2^{m+1} &= f && \text{in } \Omega_2 \\ u_2^{m+1} &= 0 && \text{on } \partial\Omega_2 \cap \partial\Omega \\ u_2^{m+1} &= u_1^{m+1} && \text{on } \Omega \setminus \Omega_2 \end{aligned}$$

By its very nature, this approach is not parallel since the solution of Poisson's equation on subdomain Ω_2 at iteration $m+1$ depends on the solution on subdomain Ω_1 at the same iteration. However, a slight modification of the original Schwarz method as studied at the continuous level by Lions [1988] yields a fully parallel algorithm. Given a first couple of iterates (u_1^0, u_2^0), solve:

$$\begin{aligned} -\Delta u_1^{m+1} &= f && \text{in } \Omega_1 \\ u_1^{m+1} &= 0 && \text{on } \partial\Omega_1 \cap \partial\Omega \\ u_1^{m+1} &= u_2^m && \text{on } \Omega \setminus \Omega_1 \end{aligned} \qquad \begin{aligned} -\Delta u_2^{m+1} &= f && \text{in } \Omega_2 \\ u_2^{m+1} &= 0 && \text{on } \partial\Omega_2 \cap \partial\Omega \\ u_2^{m+1} &= u_1^{m} && \text{on } \Omega \setminus \Omega_2 \end{aligned}$$

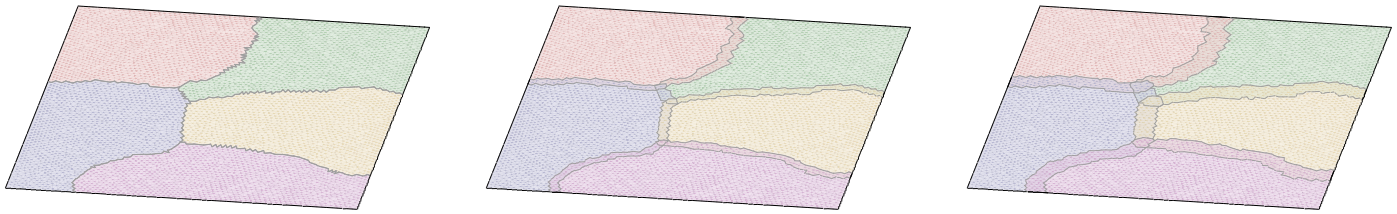
This algorithm plays a fundamental role in the rest of this section. It will first be extended to the case when there are more than two subdomains, and its algebraic formulation will be given afterwards.

Algebraic tools

The previous example will now be extended to the case of many subdomains. For that reason, it is important to introduce some algebraic notations that will be used throughout this section.
The decomposition, with one level of overlap $\left\{\mathcal{T}_i^1\right\}_{i=1}^N$ is obtained by including all the adjacent elements of $\mathcal{T}_i$, i.e. all elements of $\mathcal{T}$ sharing at least one vertex, one edge or one face (in $\mathbb{R}^3$) with at least one element of $\mathcal{T}_i$, into $\mathcal{T}_i^1$. Recursively, the decomposition with l levels of overlap $\left\{\mathcal{T}_i^l\right\}_{i=1}^N$ can be defined, and the subdomains $\left\{\Omega_i^l\right\}_{i=1}^N$ accordingly. All sets of neighboring subdomains $\{\mathcal{O}_i\}_{i=1}^N$ defined eq. (1.4) are extended to include indices of overlapping neighboring subdomains, i.e. $\forall i \in [\![1; N]\!]$,

$$\mathcal{O}_i = \mathcal{O}_i \bigcup_{m=1}^{l-1} \{j \in [\![1; N]\!] \ : \ \text{at least one element of } \mathcal{T}_j^m \text{ touches } \mathcal{T}_i^m\}.$$

Figure 1.2 is a simple example of such decompositions for a two-dimensional domain. Note that the width of the overlap is of characteristic size $2l$ elements.



(a) Original partition **(b)** One level of overlap **(c)** Two levels of overlap

Fig 1.2: Decomposition of $\Omega = [0;1]^2$ into $N = 5$ color-coded subdomains with different levels of overlap.

During the recursion, it is possible to define piecewise linear functions that will be useful afterwards. Let $\left\{\widetilde{\chi}_i^l\right\}_{i=1}^N$ be functions of $\left\{\Omega_i^l\right\}_{i=1}^N$ defined as such:

$$\widetilde{\chi}_i^l = \begin{cases} 1 & \text{on all nodes of } \Omega_i^0 \\ 1 - \dfrac{m}{l} & \text{on all nodes of } \Omega_i^l \setminus \overline{\Omega}_i^{l-1} \end{cases} \quad \forall m \in [\![1;l]\!]\,, \tag{1.6}$$

For clarity, the number of levels of overlap for the decomposition is considered in this section to be a fixed positive integer l, and the superscript ^{l} will be omitted.
On each Ω_i, the finite element space V_i is built using the same basis functions as in eq. (1.2) which support intersects Ω_i. A basis function ϕ_k is associated either with an interior degree of freedom (d.o.f.) of subdomain i if $supp(\phi_k) \subset \overline{\Omega}_i$ or to a boundary d.o.f. otherwise. Note that for all boundary d.o.f., the local function $\widetilde{\chi}_i$ evaluates to 0. The local number of d.o.f. of each V_i will be referred to as n_i.

Definition 1.1 (restriction operators). *In order to write algorithms that act on global solution vectors in V, restriction operators $\{R_i\}_{i=1}^N$ from global functions in V to local functions in $\{V_i\}_{i=1}^N$ are needed. They are, from a discrete point of view, rectangular matrices of size $n_i \times n$ filled with zeros and a single nonzero coefficient equal to one per row. Their dual operators, i.e. the extension operators from functions in $\{V_i\}_{i=1}^N$ to V, are simply $\left\{R_i^T\right\}_{i=1}^N$. Using eq. (1.2) and identifying once again a finite element function and its vector of degrees of freedom., the following algebraic definition can be given:*

$$\forall (u,i) \in \mathbb{R}^n \times [\![1;N]\!],\ R_i u = \sum_{j\,:\,int(supp(\phi_j))\cap\Omega_i\neq\emptyset} u[j]\phi_j = \sum_{j=1}^{n_i} u_j \varepsilon_j^{(i)} \in \mathbb{R}^{n_i}\,, \tag{1.7}$$

where $\forall i \in [\![1;N]\!]$, the set $\left\{\varepsilon_j^{(i)}\right\}_{j=1}^{n_i}$ is the canonical basis of $\mathbb{R}^{n_i}$.

The matrices $\{R_i\}_{i=1}^N$ are used to fulfill a very specific role: transfer information between subdomains. They can be used to define additional tools.

Definition 1.2 (numbering). *Let $\{\mathcal{N}_i\}_{i=1}^N$ be defined as:*

$$\forall i \in [\![1;N]\!],\ \mathcal{N}_i \text{ is the set (of size } n_i) \text{ of indices of nonzero columns in } R_i. \tag{1.8}$$

Let $(i,j) \in [\![1;N]\!]^2$. Then if $j \notin \overline{\mathcal{O}}_i$, $\mathcal{N}_i \cap \mathcal{N}_j = \emptyset$. Moreover, eq. (1.7) can be restated using the following formulae:

$$\begin{aligned} \forall (u,i) \in \mathbb{R}^n \times [\![1;N]\!],\ R_i u &= \sum_{j \in \mathcal{N}_i} u_j \varepsilon^{(i)}_{\mathcal{N}_i^{-1}(j)} \in \mathbb{R}^{n_i}\,, \\ \forall i \in [\![1;N]\!], \forall u^{(i)} \in \mathbb{R}^{n_i},\ R_i^T u^{(i)} &= \sum_{j=1}^{n_i} u^{(i)}_j \varepsilon_{\mathcal{N}_i(j)}\,, \\ &= \sum_{j \in \mathcal{N}_i} u^{(i)}_{\mathcal{N}_i^{-1}(j)} \varepsilon_j \in \mathbb{R}^n\,. \end{aligned} \tag{1.9}$$

Definition 1.3 (partition of unity). *A set of square diagonal matrices $\{D_i \in \mathbb{R}^{n_i \times n_i}\}_{i=1}^N$ is a partition of unity if the following equality holds:*

$$\sum_{i=1}^N R_i^T D_i R_i = I \in \mathbb{R}^{n \times n}\,. \tag{1.10}$$

Such a set can be algebraically defined as follows:

$$\forall i \in [\![1;N]\!],\ (D_i)_{jk \substack{1 \leqslant j \leqslant n_i \\ 1 \leqslant k \leqslant n_i}} = \begin{cases} 1/\#\{l \in \overline{\mathcal{O}}_i\ :\ \mathcal{N}_i(j) \in \mathcal{N}_l\} & \text{if } j = k \\ 0 & \text{otherwise.} \end{cases}$$

Based on eq. (1.6), it is possible to give another construction for the diagonal matrices $\{D_i\}_{i=1}^N$. Let $\{\widetilde{X}^{(i)}\}_{i=1}^N$ be the finite element interpolations of $\{\widetilde{\chi}_i\}_{i=1}^N$ from the space of local piecewise linear functions to each $\{V_i\}_{i=1}^N$. Then, define locally $\left\{X^{(i)}\right\}_{i=1}^N$ which are functions of each $\{V_i\}_{i=1}^N$ such that:

$$\forall i \in [\![1;N]\!],\ X^{(i)} = \frac{\widetilde{X}_i}{\sum_{j=1}^N \widetilde{X}^{(j)}_{|\Omega_i \cap \Omega_j}}\,.$$

It is straightforward to verify that:

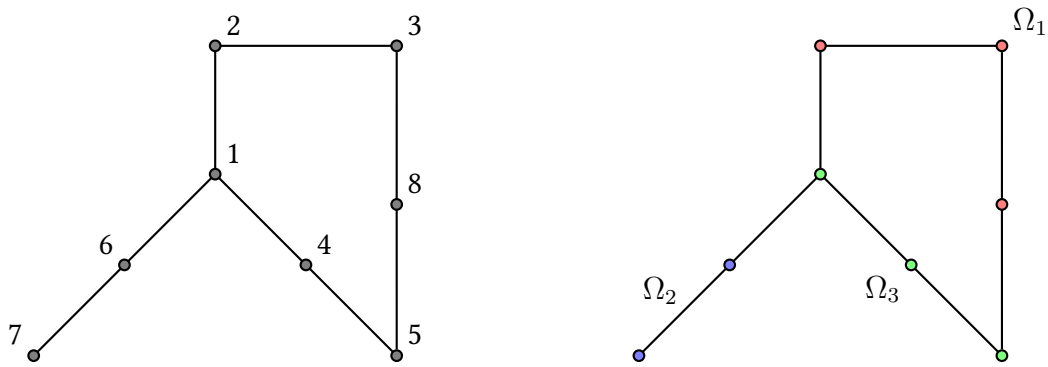
$$\sum_{i=1}^N X^{(i)} = 1\,,$$

therefore, the diagonal matrices $\{D_i\}_{i=1}^N$ defined as follows verify eq. (1.10):

$$\forall i \in [\![1;N]\!],\ (D_i)_{jk \substack{1 \leqslant j \leqslant n_i \\ 1 \leqslant k \leqslant n_i}} = \begin{cases} X^{(i)}[j] & \text{if } j = k \\ 0 & \text{otherwise.} \end{cases}$$

For exotic finite elements, the discretized partition of unity given below might not lead to an analytical partition of unity.

A short example of such operators is given below for a two-dimensional domain discretized with line segments. Nodal continuous piecewise linear finite elements and a decomposition into three subdomains with one level of overlap are now considered, so the first step is to number each node of the original mesh and partition it. The finite element nodes are identified by each end points of the line segments in figs. 1.3 to 1.4.

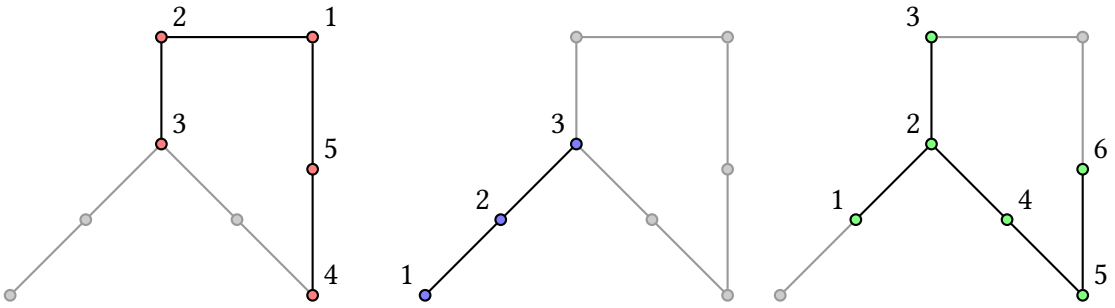


(a) Node numbering of V **(b)** Nonoverlapping decomposition

Fig 1.3: Initial numbering of the finite element space V and decomposition of Ω into $N = 3$ subdomains.

It is now possible to:

a. build the overlapping decomposition as previously and create the numbering of each local finite element space V_i:



(a) Node numbering of V_1 **(b)** Node numbering of V_2 **(c)** Node numbering of V_3

Fig 1.4: Local numbering of each local finite element space.

b. assemble the restriction matrices $\{R_i\}_{i=1}^N$ and the partition of unity $\{D_i\}_{i=1}^N$ concurrently:

$$R_1 = \begin{bmatrix} 0&0&1&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 1&0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0&1 \end{bmatrix} \qquad R_2 = \begin{bmatrix} 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&1&0&0\\ 1&0&0&0&0&0&0&0 \end{bmatrix} \qquad R_3 = \begin{bmatrix} 0&0&0&0&0&1&0&0\\ 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0&1 \end{bmatrix}.$$

c. create the sets $\{\mathcal{N}_i\}_{i=1}^N$:

$$\mathcal{N}_1 = \{3,2,1,5,8\} \qquad \mathcal{N}_2 = \{7,6,1\} \qquad \mathcal{N}_3 = \{6,1,2,4,5,8\}\,.$$

Definition 1.4. *Let $(i,j) \in [\![1;N]\!]^2$. Then define the rectangular or square matrices*

$$A_{ij} = R_i A R_j^T \in \mathbb{R}^{n_i} \times \mathbb{R}^{n_j}\,. \tag{1.11a}$$

Going back to the definition of A in eq. (1.3), these matrices can be computed as follows:

$$(A_{ij})_{kl\,\substack{1\leqslant k\leqslant n_i\\ 1\leqslant l\leqslant n_j}} = \int_\Omega a(\phi_{\mathcal{N}_j(l)}, \phi_{\mathcal{N}_i(k)})\,. \tag{1.11b}$$

If A is symmetric positive definite, then so are all $\{A_{ii}\}_{i=1}^N$.

Proposition 1.1. *Let* $(i,j) \in [\![1;N]\!]^2$. *Then,*

$$\text{if } j \notin \overline{\mathcal{O}}_i,\ R_i R_j^T = 0.$$

Proof. Looking at each coefficient of the matrix-matrix product $R_i R_j^T$, one has:

$$\begin{aligned}(R_i R_j^T)_{kl}{}_{\substack{1\leqslant k\leqslant n_i\\1\leqslant l\leqslant n_j}} &= \sum_{m=1}^{n} (R_i)_{km} \left(R_j^T\right)_{ml} \\ &= \sum_{m=1}^{n} (R_i)_{km} (R_j)_{lm} \\ &= \sum_{m\in\mathcal{N}_i\cap\mathcal{N}_j} (R_i)_{km} (R_j)_{lm}\end{aligned}$$

Then $j \notin \overline{\mathcal{O}}_i \implies (R_i R_j^T)_{kl}{}_{\substack{1\leqslant k\leqslant n_i\\1\leqslant l\leqslant n_j}} = 0$, i.e. $\mathcal{N}_i \cap \mathcal{N}_j = \emptyset$. □

1.1.1 Towards an efficient formulation for distributed computing

It is proved in this section how to compute:

- the matrix-vector product Au without explicitly building the complete linear system A,
- the dot product $(v,u) = \sum_{i=1}^{n} v_i u_i$ in parallel.

All vectors $\left\{u^{(i)}\right\}_{i=1}^{N}$ are supposed to be stored in a local contiguous block of memory. In total, storage for $\sum_{i=1}^{N} n_i$ scalars must be allocated, which is larger than the size n of the original system. Using the mathematical formalism, duplicated unknowns will have the same value across processes which could then be accessed without any interprocess communications. These algebraic operations are key in an iterative method, because they could be called multiple times per iteration. Avoiding the explicit construction of the complete system A can be quite useful if a decomposition similar to fig. 1.4 is used, and that it is assumed that local meshes and finite element spaces are distributed among a group of processes. In that case, each process $i \in [\![1;N]\!]$ can only assemble easily its own diagonal block A_{ii}.

Proposition 1.2. *Let* $(u,i) \in \mathbb{R}^n \times [\![1;N]\!]$. *Then,*

$$R_i^T R_i u = u \iff \forall j \in [\![1;n]\!] \setminus \mathcal{N}_i, u_j = 0\,.$$

Proof. Using eq. (1.9) for $(u,i) \in \mathbb{R}^n \times [\![1;N]\!]$, one has:

$$R_i^T R_i u = \sum_{j\in\mathcal{N}_i} u_j \varepsilon_j \in \mathbb{R}^n\,,$$

so that $R_i^T R_i u = \sum_{j=1}^{n} u_j \varepsilon_j \iff u_j = 0, \forall j \in [\![1;n]\!] \setminus \mathcal{N}_i$. □

Proposition 1.3. *Let* $(u,i) \in \mathbb{R}^n \times [\![1;N]\!]$. *Then,*

$$\forall j \in [\![1;n]\!] \setminus \mathcal{N}_i,\ \left(A R_i^T D_i R_i u\right)_j = 0\,.$$

Proof. Using eq. (1.9) for $(u,i) \in \mathbb{R}^n \times [\![1;N]\!]$ and the fact that diagonal entries of D_i associated with boundary d.o.f. are equal to 0 one has:

$$\forall k \in [\![1;n]\!],\ (R_i^T D_i R_i u)_k \neq 0 \implies \text{supp}(\phi_k) \subset \overline{\Omega}_i\,,$$

and because $\forall (j,k) \in [\![1;n]\!]^2, A_{jk} \neq 0 \implies \text{int}\,(\text{supp}(\phi_j)) \cap \text{int}\,(\text{supp}(\phi_k)) \neq \emptyset$, one can infer that:

$$\forall j \in [\![1;n]\!],\ \text{int}\,(\text{supp}(\phi_j)) \cap \underbrace{\left(\bigcup_{k\,:\,\text{supp}(\phi_k)\subset\overline{\Omega}_i} \text{supp}(\phi_k) \right)}_{=\overline{\Omega}_i} = \emptyset \implies (AR_i^T D_i R_i u)_j = 0\,.$$

By definition, $\text{int}\,(\text{supp}(\phi_j)) \cap \Omega_i = \emptyset \iff j \notin \mathcal{N}_i$, cf. eq. (1.7). □

As a consequence of propositions 1.2 and 1.3, the following theorem is the cornerstone of the matrix-vector product.

Theorem 1.4. *Let $u \in \mathbb{R}^n$. Then,*

$$\begin{aligned} Au &= \sum_{i=1}^{N} R_i^T R_i A R_i^T D_i R_i u \\ &= \sum_{i=1}^{N} R_i^T A_{ii} D_i R_i u\,. \end{aligned}$$

Proof. Using eq. (1.10),

$$Au = \sum_{i=1}^{N} A R_i^T D_i R_i u\,.$$

Given $i \in [\![1;N]\!]$, applying prop. 1.3 to the ith term of the previous sum leads to:

$$\forall j \in [\![1;n]\!] \setminus \mathcal{N}_i,\ (AR_i^T D_i R_i u)_j = 0,$$

such that, using prop. 1.2 gives:

$$R_i^T R_i A R_i^T D_i R_i u = A R_i^T D_i R_i u\,.$$

The previous equality being true for all $i \in [\![1;N]\!]$:

$$\sum_{i=1}^{N} R_i^T R_i A R_i^T D_i R_i u = \sum_{i=1}^{N} A R_i^T D_i R_i u\,. \quad \square$$

It is now possible to compute the matrix-vector product without the off-diagonal blocks of A. This might seem counterintuitive at first, but can be explained by the fact that unknowns are duplicated in the overlap, so that part of local stiffness matrices $\{A_{ii}\}_{i=1}^N$ are also duplicated and this can be used in conjunction with the partition of unity.
Because it is assumed that the local finite element spaces are distributed among processes, there is actually no need to compute the complete vector Au. Instead, its restriction to each local finite element space is given by:

$$\begin{aligned} \forall i \in [\![1;N]\!],\ R_i Au &= R_i \sum_{j=1}^{N} R_j^T A_{jj} D_j R_j u \\ &= \sum_{j\in\overline{\mathcal{O}}_i} R_i R_j^T A_{jj} D_j R_j u \qquad \text{using proposition 1.1.} \end{aligned} \tag{1.12}$$

In eq. (1.12), there are only local operations, except the matrix-matrix products $\left\{R_i R_j^T\right\}_{\substack{1 \leqslant i \leqslant N \\ j \in \overline{\mathcal{O}}_i}}$. These operators are used to exchange information on the overlap between subdomains, and as a consequence, for $i \in [\![1; N]\!]$ and $j \in \overline{\mathcal{O}}_i$, the action of $R_i R_j^T$ on a vector $u^{(j)} \in \mathbb{R}^{n_j}$ can be computed as:

$$R_i R_j^T u^{(j)} = \sum_{k \in \mathcal{N}_i \cap \mathcal{N}_j} u^{(j)}_{\mathcal{N}_j^{-1}(k)} \varepsilon_{\mathcal{N}_i^{-1}(k)} \in \mathbb{R}^{n_i} . \tag{1.13}$$

A satisfying property of eq. (1.13) is that it does not involve any global information: there is no need for a numbering of the finite element space V as done fig. 1.3. This is convenient when there is no actual way to compute the global numbering, when the finite element kernel is sequential or used as a black box and can only provide elementary finite element matrices.

Theorem 1.5. *Let $(u, v) \in \mathbb{R}^n \times \mathbb{R}^n$. Then,*

$$(u, v) = \sum_{i=1}^{N} (R_i u, D_i R_i v) .$$

Proof. Using eq. (1.10), this is a direct consequence because:

$$\begin{aligned} (u, v) = \left(u, \sum_{i=1}^{N} R_i^T D_i R_i v \right) &= \sum_{i=1}^{N} (R_i u, D_i R_i v) \\ &= \sum_{i=1}^{N} \left(u^{(i)}, D_i v^{(i)} \right) . \end{aligned}$$ □

This time, theorem 1.5 displays how it is possible to compute a dot product using local dot products followed by a summation over all subdomains.

1.1.2 One-level methods

Two preconditioners will now be introduced to solve eq. (1.3). They are based on the idea illustrated in the introductory paragraph of this section, which is itself based on the "divide & conquer" paradigm: for solving one very large sparse linear system, it might be adequate to use a preconditioner using N smaller linear systems. Once again, it will be stressed how it is possible to avoid global information so that it is feasible to work on simple data structures.

Additive Schwarz method

The Additive Schwarz Method (ASM) is a preconditioner used for the following fixed-point iteration:

$$u^{m+1} = u^m + M_{\text{ASM}}^{-1}(f - Au^m) ,$$

where M_{ASM}^{-1} is defined as the sum of all local solvers. That is,

$$M_{\text{ASM}}^{-1} = \sum_{i=1}^{N} R_i^T {A_{ii}}^{-1} R_i .$$

If the matrix A is symmetric, then M_{ASM}^{-1} is also symmetric. Restricted to a single subdomain, the left multiplication of a vector $u \in \mathbb{R}^n$ by this preconditioner yields for $i \in [\![1; N]\!]$:

$$R_i M_{\text{ASM}}^{-1} u = R_i \sum_{j \in \overline{\mathcal{O}_i}} R_j^T {A_{jj}}^{-1} u^{(j)} . \tag{1.14}$$

Note that there is no need for global information, only local and transfer operators $\{A_{ii}\}_{i=1}^{N}$, $\{R_i R_j^T\}_{\substack{1\leqslant i\leqslant N \\ j\in\overline{\mathcal{O}}_i}}$.

If the subdomains are of characteristic size H, and if the level of overlap is kept constant, it was shown by Le Tallec [1994] that the condition number of the preconditioner system grows as $\frac{1}{H}$. This means that, for a given domain Ω, if the number of subdomains in the decomposition increases, their characteristic size will decrease and the condition number will increase. In the end, an iterative solver preconditioned by such a method will not scale to large numbers of subdomains because its number of iterations to reach a desired accuracy will increase.

Restricted additive Schwarz method

The Restricted Additive Schwarz method (RAS) is a variant of ASM which was introduced by Cai and Sarkis [1999] and is then used for the following fixed-point iteration:

$$u^{m+1} = u^m + M_{\text{RAS}}^{-1}(f - Au^m)\,,$$

where M_{RAS}^{-1} is defined as the sum of all local solvers weighted by some partition of unity. That is,

$$M_{\text{RAS}}^{-1} = \sum_{i=1}^{N} R_i^T D_i {A_{ii}}^{-1} R_i\,.$$

Even if the matrix A is symmetric, the preconditioner M_{RAS}^{-1} is not symmetric. Restricted to a single subdomain, the left multiplication of a vector $u \in \mathbb{R}^n$ by this preconditioner yields for $i \in [\![1; N]\!]$ a formulation close to eq. (1.14):

$$R_i u = R_i \sum_{j\in\overline{\mathcal{O}}_i} R_j^T D_j {A_{jj}}^{-1} u^{(j)}\,. \tag{1.15}$$

Further extensions of RAS have been made, see for example [Cai, Dryja, and Sarkis 2003].

1.1.3 Two-level methods

By looking at eqs. (1.14) and (1.15), it is clear that at each iteration, a subdomain i only communicates information to its nearest neighbors $\mathcal{O}_i$. This lack of global exchange between all subdomains explains the condition number of one-level methods that does not scale as the number of subdomains N increases: the high-frequency errors can be reduced quickly but low eigenvalues in the preconditioned systems hamper the reduction of low-frequency errors. The goal is then to introduce an additional operator that will deal with these low eigenvalues. The way a second level is added to a basic one-level preconditioner is a standard approach that has many similarities with the field of multigrid methods, see for example [Vassilevski 2008].

Definition 1.5 (Galerkin operator). *Let Z be a full rank, tall and skinny matrix of dimensions $n \times m$ ($m \ll n$). Then, define the Galerkin operator[2] E as:*

$$E = Z^T A Z \in \mathbb{R}^{m\times m}\,. \tag{1.16}$$

If A is symmetric positive definite (SPD), then so is E.

[2] Also referred to as RAP (Restriction Prolongation) in the multigrid community.

Because E is of lower dimension than A, it will be referred to as a coarse operator. It is used to enrich a one-level method M^{-1}, for example additively:

$$P_{\text{AD}}^{-1} = ZE^{-1}Z^T + M^{-1}\,. \tag{1.17}$$

In section 1.3, an automatic and efficient way to build the deflation subspace matrix Z will be presented, but a simple approach is given here first. It was introduced by Nicolaides [1987] where the author chose Z as:

$$\forall (i,j) \in \mathbb{R}^n \times \mathbb{R}^N,\ Z_{ij} = \begin{cases} (D_j)_{\mathcal{N}_j^{-1}(i)\mathcal{N}_j^{-1}(i)} & \text{if } i \in \mathcal{N}_j \\ 0 & \text{otherwise.} \end{cases} \tag{1.18}$$

If on each subdomain $j \in [\![1; N]\!]$, a constant vector $\mathbb{1}_i$ equal to 1 is defined on V_i, then Z is nothing else than the column-wise concatenation of all $\{R_i^T \mathbb{1}_i\}_{i=1}^N$. This works well in the case of Poisson's equation, as this time, unlike one-level methods, P_{AD}^{-1} theoretically scales with increasing numbers of subdomains—for a proof, see [Dolean, Jolivet, and Nataf 2014]. Other deflation subspace matrices have been introduced, for example by Sarkis [2003], that make two-level methods for systems of PDEs, such as the equations of linear elasticity, scale as well.
Eventually, once the coarse operator is built, there is more than one way to enrich one-level preconditioners as done eq. (1.17). Tang et al. [2009] compare various coarse operator corrections. It is theoretically and numerically shown that the following preconditioners share some properties with P_{AD}^{-1}:

$$P_{\text{A-DEF1}}^{-1} = M^{-1}(I - AQ) + Q \tag{1.19a}$$
$$P_{\text{A-DEF2}}^{-1} = (I - QA)M^{-1} + Q \tag{1.19b}$$
$$P_{\text{BNN}}^{-1} = (I - QA)M^{-1}(I - AQ) + Q\,, \tag{1.19c}$$

where $Q = ZE^{-1}Z^T$. A comparison of the performance of these operators is available in [Jolivet, Dolean, et al. 2012].

For a more in-depth introduction to overlapping Schwarz methods, the reader is referred to [Mathew 2008; Quarteroni and Valli 1999; Smith, Bjørstad, and Gropp 2004; Toselli and Widlund 2005].

1.2 Substructuring methods

While overlapping methods described in the previous section are purely algebraic and can precondition eq. (1.3) efficiently, they have some drawbacks:

1. due to the duplication of unknowns on the overlap, some redundant work is needed, and the amount of communication may be high (the greater the overlap level, the more problematic this can be),

2. when solving a coupled multiphysics problem, a natural way to partition the computational domain Ω would be to split each physics into a subdomain, but that is not possible with overlapping subdomains,

3. when solving a heterogeneous problem, for example the equations of elasticity with multiple mechanical parts having different elastic properties, a natural way to partition Ω would be to split each part into a subdomain, but the same limitation appears,

4. finally, even in the case of simple and homogeneous problems, building the overlapping decomposition itself is not an easy task, since routines for manipulating meshes are needed, cf. fig. 1.2.

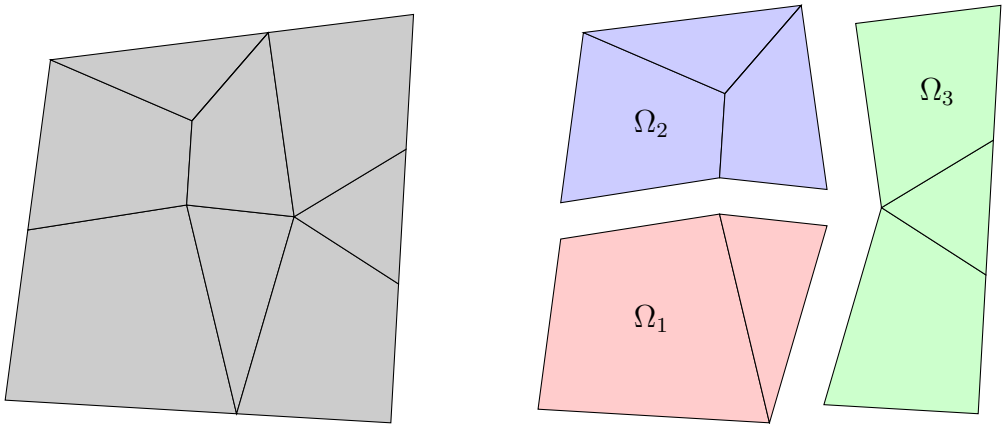
For these reasons, in the beginning of the 1990s, researchers—mainly in mechanical engineering—have been working on nonoverlapping domain decomposition (or substructuring) methods, see for example [De Roeck and Le Tallec 1991]. Modern introductions to substructuring methods can be found in [Boiteau 2009; Gosselet and Rey 2006; Klawonn and Rheinbach 2012; Pechstein 2012; Rheinbach 2009]. In section 1.2.1, a so-called primal approach will be described: the Balancing Neumann-Neumann method [Mandel 1993]. In section 1.2.2, a so-called dual approach will be described: the Finite Element Tearing and Interconnecting (FETI) method [Farhat and Roux 1991]. However, both approaches share some numerical tools that will now be described in this preliminary section, which is greatly inspired by [Gosselet and Rey 2006]. The same notations as in section 1.1 will be used, in particular, the definitions of $\{R_i\}_{i=1}^N$ and $\{\mathcal{N}_i\}_{i=1}^N$ are extended to the case when the level of overlap is null ($l = 0$). Moreover,

- all sets $\{\mathcal{N}_j\}_{j=1}^N$ are split into sets of interior $\left\{\mathcal{N}_i^{(j)}\right\}_{j=1}^N$ and boundary $\left\{\mathcal{N}_b^{(j)}\right\}_{j=1}^N$ d.o.f. indices,
- the number of boundary (resp. interior) d.o.f. of subdomain $j \in [\![1; N]\!]$ will be referred to as $n_b^{(j)}$ (resp. $n_i^{(j)}$) so that $n_j = n_i^{(j)} + n_b^{(j)} = \#\mathcal{N}_i^{(j)} + \#\mathcal{N}_b^{(j)}$,
- the set of all boundary d.o.f. will be represented as $\mathcal{N}_b = \bigcup_{j=1}^N \mathcal{N}_b^{(j)}$.

Definition 1.6. *Given a d.o.f. $k \in [\![1; n]\!]$, its multiplicity m_k equals:*

$$m_k = \#\left\{i \in [\![1; N]\!] \ : \ k \in \mathcal{N}_i\right\}.$$

Obviously, $\forall k \in \mathcal{N}_i$, $m_k = 1$ and $\forall k \in \mathcal{N}_b$, $m_k \geqslant 2$. If $m_k > 2$, the d.o.f. k is said to be a cross-point.



(a) Original geometry Ω **(b)** Nonoverlapping decomposition

Fig 1.5: Substructuring of Ω into $N = 3$ subdomains.

Definition 1.7 (unassembled operators)**.** *Unlike overlapping methods which use assembled operators $\{A_{ii}\}_{i=1}^N$, cf. eq. (1.11), substructuring methods require unassembled operators $\left\{\mathring{A}_i\right\}_{i=1}^N$:*

$$\left(\mathring{A}_i\right)_{jk \substack{1 \leqslant j \leqslant n_i \\ 1 \leqslant k \leqslant n_i}} = \int_{\Omega_i} a(\phi_{\mathcal{N}_i(k)}, \phi_{\mathcal{N}_i(j)}) \, . \tag{1.20}$$

They are symmetric positive semidefinite. In a similar way, $\left\{\mathring{f}_i\right\}_{i=1}^N$ *are defined as:*

$$\left(\mathring{f}_i\right)_{j_1\leqslant j\leqslant n_i} = \int_{\Omega_i} l(\phi_{\mathcal{N}_i(k)})\,.$$

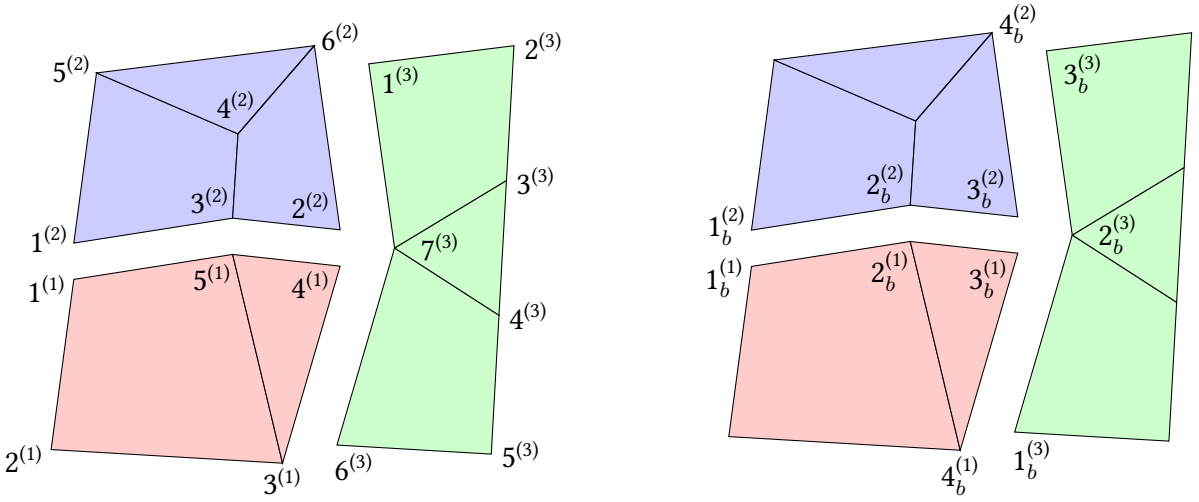
Proposition 1.6. *To better understand the link between assembled and unassembled operators, the following equalities can be useful:*

$$A = \sum_{i=1}^N R_i^T \mathring{A}_i R_i \qquad\qquad f = \sum_{i=1}^N R_i^T \mathring{f}_i\,.$$

Definition 1.8 (trace operators)**.** *Let* $\{W_i\}_{i=1}^N$ *be the finite element spaces associated with the interfaces* $\{\partial\Omega\cap\partial\Omega_i\}_{i=1}^N$*. In addition to restriction operators that map global functions in* V *to local functions in* $\{V_i\}_{i=1}^N$*, trace operators* $\left\{t^{(i)}\right\}_{i=1}^N$ *are needed for functions from* $\{V_i\}_{i=1}^N$ *to* $\{W_i\}_{i=1}^N$*. Algebraically,*

$$\forall i\in[\![1;N]\!], \forall u^{(i)}\in V_i,\ t^{(i)}u_i = \sum_{k\in\mathcal{N}_b^{(i)}} u^{(i)}_{\mathcal{N}_i^{-1}(k)}\varepsilon_{\mathcal{N}_i^{-1}(k)} \in \mathbb{R}^{n_b^{(i)}}\,.$$

As for the restriction operators, $\left\{t^{(i)^T}\right\}_{i=1}^N$ *will be used to extend by 0 trace functions on* $\{V_i\}_{i=1}^N$*.*



(a) Node numbering of each $\{V_i\}_{i=1}^N$

(b) Node numbering of each $\{W_i\}_{i=1}^N$

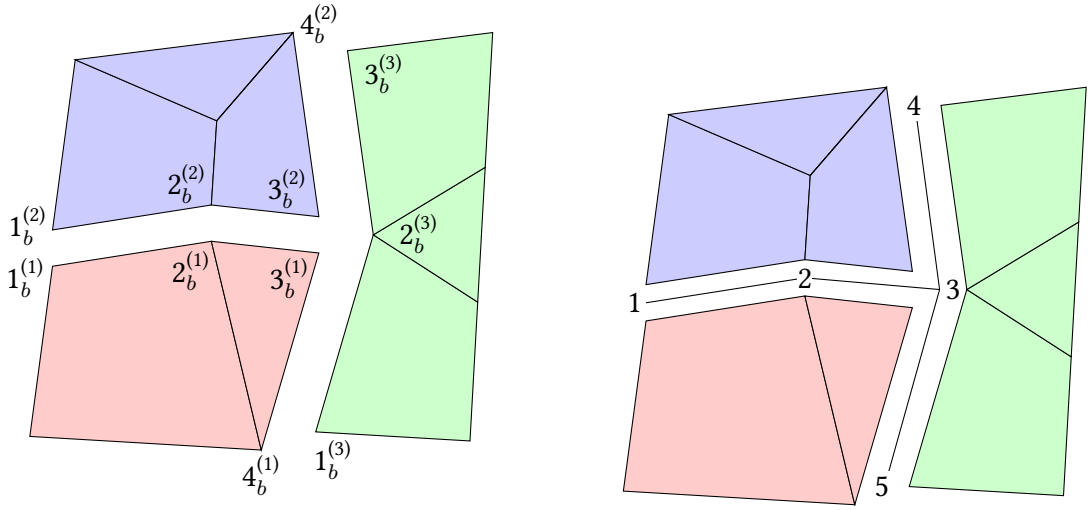
$$t^{(1)} = \begin{bmatrix} 1&0&0&0&0\\ 0&0&0&0&1\\ 0&0&0&1&0\\ 0&0&1&0&0 \end{bmatrix} \qquad t^{(2)} = \begin{bmatrix} 1&0&0&0&0&0\\ 0&0&1&0&0&0\\ 0&1&0&0&0&0\\ 0&0&0&0&0&1 \end{bmatrix} \qquad t^{(3)} = \begin{bmatrix} 0&0&0&0&0&1&0\\ 0&0&0&0&0&0&1\\ 1&0&0&0&0&0&0 \end{bmatrix}.$$

Fig 1.6: Trace operators $\left\{t^{(i)}\right\}_{i=1}^N$.

The goal of substructuring methods is to eliminate the interior d.o.f. Hence, subdomains will only exchange information through their interfaces. For that reason, assembly operators and jump operators, that will play a similar role to restriction operators in the case of overlapping Schwarz methods, have to be introduced.

Definition 1.9. *Primal assembly operators are defined as follows:*

$$\forall i \in [\![1;N]\!],\ \left(B^{(i)}\right)_{jk\substack{1\leqslant j\leqslant \#\mathcal{N}_b \\ 1\leqslant k\leqslant n_b^{(i)}}} = \begin{cases} 1 & \text{if } \mathcal{N}_b^{(i)}(k) = j \\ 0 & \text{otherwise.} \end{cases}$$



(a) Node numbering of each $\{W_i\}_{i=1}^N$ **(b)** Node numbering of the interface

$$B^{(1)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad B^{(2)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad B^{(3)} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

Fig 1.7: Primal assembly operators $\left\{B^{(i)}\right\}_{i=1}^N$.

Imposing constraints across each subdomain interface will be needed for substructuring methods. In particular, the solution u of eq. (1.3) must be continuous at the interface. This can be written as:

$$\forall (i,j) \in [\![1;N]\!], i \neq j, \forall k \in \mathcal{N}_b^{(i)} \cap \mathcal{N}_b^{(j)}, u^{(i)}_{\mathcal{N}_i^{-1}(k)} = u^{(j)}_{\mathcal{N}_j^{-1}(k)}\,. \tag{1.21a}$$

Some of these equality constraints are redundant in the presence of cross-points, but this can be avoided. Let $\mathcal{S}_k = \left\{i \in [\![1;N]\!] \ :\ k \in \mathcal{N}_i\right\}$ then:

$$\forall k \in \mathcal{N}_b, \forall i \in \mathcal{S}_k \setminus \{\max \mathcal{S}_k\}, j = \mathcal{S}_k\left(\mathcal{S}_k^{-1}(i)+1\right) \text{ and } u^{(i)}_{\mathcal{N}_i^{-1}(k)} = u^{(j)}_{\mathcal{N}_j^{-1}(k)}\,. \tag{1.21b}$$

The idea behind the previous equation is to impose, at cross-points, equality constraints only between neighboring subdomains with successive indices. Let M be the number of constraints imposed by either eqs. (1.21a) or (1.21b) and Λ be an index mapping from $[\![1;N]\!]^2 \times \mathcal{N}_b$ onto $[\![0;M]\!]$—value 0 being reserved for when there is no constraint imposed by the input triplet $(i,j,k) \in [\![1;N]\!]^2 \times \mathcal{N}_b$, for example when $j \notin \overline{\mathcal{O}}_i$.

Proposition 1.7. *In the case of eq.* (1.21a),

$$M = \frac{\displaystyle\sum_{i=1}^{N} \sum_{k\in\mathcal{N}_b^{(i)}} (m_k - 1)}{2} = \sum_{k\in\mathcal{N}_b} \frac{m_k(m_k-1)}{2}\,,$$

and in the case of eq. (1.21b),

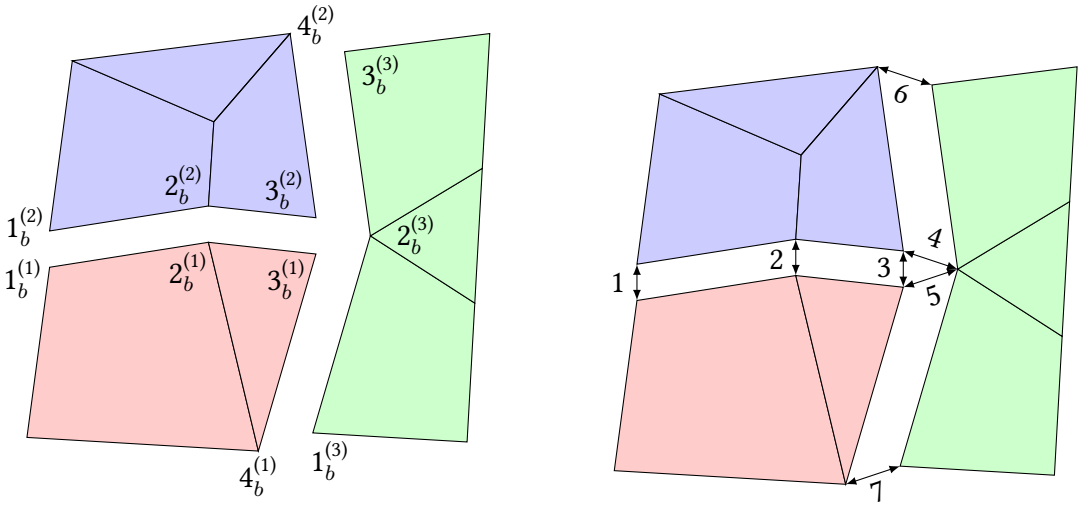
$$M = \sum_{k \in \mathcal{N}_b} (m_k - 1)\,.$$

Definition 1.10. *Dual jump operators are defined as follows and are used to represent in matrix form the set of constraints eq.* (1.21)*:*

$$\forall i \in [\![1; N]\!],\ \left(\underline{B}^{(i)}\right)_{kl\ \substack{1\leqslant k\leqslant M \\ 1\leqslant l\leqslant n_b^{(i)}}} = \begin{cases} 1 & \text{if } \exists j \in \mathcal{O}_i \,:\, j > i \text{ and } \Lambda\left(i, j, \mathcal{N}_b^{(i)}(l)\right) = k \\ -1 & \text{if } \exists j \in \mathcal{O}_i \,:\, j < i \text{ and } \Lambda\left(i, j, \mathcal{N}_b^{(i)}(l)\right) = k \\ 0 & \text{otherwise.} \end{cases} \tag{1.22}$$

From that definition, one can infer that:

$$\forall k \in \mathcal{N}_b, \forall i \in \mathcal{S}_k, \exists!(j, l) \in \mathcal{S}_k \setminus \{i\} \times [\![1; M]\!] \,:\, \left(\underline{B}^{(i)}\right)_{l\mathcal{N}_i^{-1}(k)} = -\left(\underline{B}^{(j)}\right)_{l\mathcal{N}_j^{-1}(k)}.$$



(a) Node numbering of each $\{W_i\}_{i=1}^N$

(b) Numbering of the set of constraints

$$\underline{B}^{(1)} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \quad \underline{B}^{(2)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \underline{B}^{(3)} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

Fig 1.8: Redundant jump operators $\left\{\underline{B}^{(i)}\right\}_{i=1}^N$ defined using eq. (1.21a).

Proposition 1.8. *The assembly and jump operators are orthogonal in the sense that:*

$$\sum_{i=1}^{N} \underline{B}^{(i)} B^{(i)^T} = 0. \tag{1.23}$$

Proof. $\forall (i, j, k) \in [\![1; N]\!] \times [\![1; M]\!] \times [\![1; \#\mathcal{N}_b]\!]$, the coefficient $b_{jk}^{(i)}$ of the matrix $\underline{B}^{(i)} B^{(i)^T}$ is:

$$b_{jk}^{(i)} = \sum_{l=1}^{\#\mathcal{N}_b^{(i)}} \left(\underline{B}^{(i)}\right)_{jl} \left(B^{(i)}\right)_{kl}.$$

Hence,

$$\sum_{i=1}^{N} b_{jk}^{(i)} = \sum_{l \in \mathcal{S}_k} \left(\underline{B}^{(l)}\right)_{j\mathcal{N}_l^{-1}(k)} = 0. \quad \square$$

In order to write equations that are local to each subdomain, one needs to introduce reaction unknowns $\{\lambda_i\}_{i=1}^N$ that are nonzero only on the boundary d.o.f. and that represent the reaction imposed by neighboring subdomains. In that case, eq. (1.3) is equivalent to:

$$\forall i \in [\![1; N]\!],\ \mathring{A}_i u_i = \mathring{f}_i + \lambda_i \tag{1.24a}$$

$$\sum_{i=1}^{N} \underline{B}^{(i)} t^{(i)} u_i = 0 \tag{1.24b}$$

$$\sum_{i=1}^{N} B^{(i)} t^{(i)} \lambda_i = 0\,. \tag{1.24c}$$

Equation (1.24a) relates to the equilibrium of each subdomain under given body and surface forces, eq. (1.24b) relates to the compatibility of the solution across interfaces and is the algebraic representation of eq. (1.21), and eq. (1.24c) relates to the equilibrium of the interface (Newton's 3rd law).
Assuming that the jth unassembled operator can be renumbered so that the first $n_i^{(j)}$ (resp. last $n_b^{(j)}$) rows and columns are associated with interior (resp. boundary) d.o.f., one can write:

$$\forall j \in [\![1; N]\!],\ \mathring{A}_j = \begin{bmatrix} \mathring{A}_{ii}^{(j)} & \mathring{A}_{ib}^{(j)} \\ \mathring{A}_{bi}^{(j)} & \mathring{A}_{bb}^{(j)} \end{bmatrix} \qquad u_j = \begin{bmatrix} u_i^{(j)} \\ u_b^{(j)} \end{bmatrix} \qquad \mathring{f}_j = \begin{bmatrix} \mathring{f}_i^{(j)} \\ \mathring{f}_b^{(j)} \end{bmatrix} \qquad \lambda_j = \begin{bmatrix} 0 \\ \lambda_b^{(j)} \end{bmatrix}.$$

Performing a Gaussian elimination on the interior d.o.f. leads to:

$$\begin{aligned} \forall j \in [\![1; N]\!],\ \left(\mathring{A}_{bb}^{(j)} - \mathring{A}_{bi}^{(j)} \mathring{A}_{ii}^{(j)^{-1}} \mathring{A}_{ib}^{(j)}\right) u_b^{(j)} &= -\mathring{A}_{bi}^{(j)} \mathring{A}_{ii}^{(j)^{-1}} \mathring{f}_i^{(j)} + \mathring{f}_b^{(j)} + \lambda_b^{(j)} \\ u_i^{(j)} &= \mathring{A}_{ii}^{(j)^{-1}} \left(\mathring{f}_i^{(j)} - \mathring{A}_{ib}^{(j)} u_b^{(j)}\right). \end{aligned} \tag{1.25}$$

Definition 1.11. *On each subdomain, the Schur complement, which is a symmetric positive semidefinite matrix, is defined as:*

$$\forall j \in [\![1; N]\!],\ S_j = \mathring{A}_{bb}^{(j)} - \mathring{A}_{bi}^{(j)} \mathring{A}_{ii}^{(j)^{-1}} \mathring{A}_{ib}^{(j)}\,,$$

and the condensed body force is defined as:

$$\forall j \in [\![1; N]\!],\ g_j = \mathring{f}_b^{(j)} - \mathring{A}_{bi}^{(j)} \mathring{A}_{ii}^{(j)^{-1}} \mathring{f}_i^{(j)}\,.$$

Substructuring methods heavily rely on these Schur complements, and for this reason, they are also known as Schur complement domain decomposition methods. Equation (1.24) can now be written:

$$\begin{aligned} \forall i \in [\![1; N]\!],\ S_i u_b^{(i)} &= g_i + \lambda_b^{(i)} \\ \sum_{i=1}^{N} \underline{B}^{(i)} u_b^{(i)} &= 0 \\ \sum_{i=1}^{N} B^{(i)} \lambda_b^{(i)} &= 0\,, \end{aligned}$$

or using block notations, introducing:

$$S = \text{diag}(S_1, \dots, S_N) \qquad u_b = \begin{bmatrix} u_b^{(1)} \\ \vdots \\ u_b^{(N)} \end{bmatrix} \qquad g = \begin{bmatrix} g_1 \\ \vdots \\ g_N \end{bmatrix} \qquad \lambda_b = \begin{bmatrix} \lambda_b^{(1)} \\ \vdots \\ \lambda_b^{(N)} \end{bmatrix}$$

$$B = \begin{bmatrix} B^{(1)} & \cdots & B^{(N)} \end{bmatrix} \qquad \underline{B} = \begin{bmatrix} \underline{B}^{(1)} & \cdots & \underline{B}^{(N)} \end{bmatrix},$$

then,

$$Su_b = g + \lambda_b \tag{1.26a}$$
$$\underline{B}u_b = 0 \tag{1.26b}$$
$$B\lambda_b = 0\,. \tag{1.26c}$$

Proposition 1.9. *In its block formulation, proposition 1.8 reads:*

$$\underline{B}B^T = 0. \tag{1.27}$$

Two preconditioners will now be described to solve eq. (1.26). The primal approach in section 1.2.1 introduces a unique interface solution $u \in \mathbb{R}^{\#\mathcal{N}_b}$ (such that $u_b = B^T u$) satisfying eq. (1.26b) and aims at finding a solution of eq. (1.26c) iteratively. The dual approach in section 1.2.2 introduces a unique interface reaction $\lambda \in \mathbb{R}^M$ (such that $\lambda_b = \underline{B}^T\lambda$) satisfying eq. (1.26c) and aims at finding a solution of eq. (1.26b) iteratively. In both approaches, eq. (1.26a) must be verified at each iteration.
Just as when formulating the overlapping Schwarz methods in section 1.1, it will be shown how to avoid the explicit construction of operators $\left\{B^{(i)}\right\}_{i=1}^N$ and $\left\{\underline{B}^{(i)}\right\}_{i=1}^N$, which essentially act as $\left\{R_i^T\right\}_{i=1}^N$ in the case of overlapping preconditioners. Besides, all vectors $\left\{u_b^{(i)}\right\}_{i=1}^N$ and $\left\{\lambda_b^{(i)}\right\}_{i=1}^N$ are supposed to be stored in local contiguous blocks of memory. In total, for example when examining the case of the primal interface solution, storage for $\sum_{i=1}^N n_b^{(i)}$ scalars must be allocated, which is at least twice the number of boundary d.o.f.[3]. Using the mathematical formalism, duplicated unknowns will have the same value across processes which could then be accessed without any interprocess communications. The same applies for the dual interface reaction, except that now, duplicated unknowns will have opposite values across processes, cf. proposition 1.8.

1.2.1 Balancing Neumann-Neumann preconditioner

This method was historically introduced in [Mandel 1993] and is also referred to as Balancing Domain Decomposition (BDD). Given a unique interface solution $u \in \mathbb{R}^{\#\mathcal{N}_b}$ such that $u_b = B^T u$, it is known, thanks to eq. (1.27), that $\underline{B}u_b = 0$, i.e. eq. (1.26b) is always verified. In eq. (1.26a), this gives $SB^T u = g + \lambda_b$ and using eq. (1.26c), the primal formulation of the interface solution is:

$$BSB^T u = Bg,$$

or alternatively,

$$\sum_{i=1}^N B^{(i)} S_i B^{(i)^T} u = \sum_{i=1}^N B^{(i)} g_i\,. \tag{1.28}$$

[3] It is exactly twice when there is no cross-point.

Once the solution on the interface is known, the solution of eq. (1.3) associated with interior d.o.f. can be computed using eq. (1.25). Restricted to the interface of a single subdomain $i \in [\![1; N]\!]$, this yields:

$$\begin{aligned} & B^{(i)^T} \sum_{j=1}^{N} B^{(j)} S_j B^{(j)^T} u = B^{(i)^T} \sum_{j=1}^{N} B^{(j)} g_j \\ \Longrightarrow \quad & \sum_{j \in \overline{\mathcal{O}}_i} B^{(i)^T} B^{(j)} S_j u_b^{(j)} = \sum_{j \in \overline{\mathcal{O}}_i} B^{(i)^T} B^{(j)} g_j \,. \end{aligned} \tag{1.29}$$

In eq. (1.29), there are only local operations, except the matrix-matrix products $\left\{ B^{(i)^T} B^{(j)} \right\}_{\substack{1 \leqslant i \leqslant N \\ j \in \overline{\mathcal{O}}_i}}$. These operators are used to exchange information on the interface between subdomains, and as a consequence, for $i \in [\![1; N]\!]$ and $j \in \overline{\mathcal{O}}_i$, the action of $B^{(i)^T} B^{(j)}$ on a vector $u^{(j)} \in \mathbb{R}^{n_b^{(j)}}$ can be computed as:

$$B^{(i)^T} B^{(j)} u^{(j)} = \sum_{k \in \mathcal{N}_b^{(i)} \cap \mathcal{N}_b^{(j)}} u^{(j)}_{\mathcal{N}_j^{-1}(k)} \varepsilon_{\mathcal{N}_i^{-1}(k)} \in \mathbb{R}^{n_b^{(i)}} . \tag{1.30}$$

A satisfying property of eq. (1.30) is that it does not involve any global information: there is no need for a numbering of the set of boundary d.o.f. $\mathcal{N}_b$.
In order to build a preconditioner for eq. (1.28), the inverse of the sum has to be efficiently approximated. The one-level Neumann-Neumann preconditioner, defined on the skeleton of the domain decomposition, is the scaled sum of the inverse of each Schur complement:

$$M^{-1} = \sum_{i=1}^{N} B^{(i)} D_i S_i^{\dagger} D_i B^{(i)^T} , \tag{1.31}$$

where:

- $\{D_i\}_{i=1}^N$ is a partition of unity, cf. definition 1.3,
- $\left\{ S_i^{\dagger} \right\}_{i=1}^{N}$ are pseudoinverses of each local Schur complement.

They are many ways to build a partition of unity in the context of substructuring preconditioners, see for example [Farhat and Roux 1994; Rixen and Farhat 1997], but the three more frequently used are:

- the multiplicity scaling:

$$\forall i \in [\![1; N]\!], \forall k \in [\![1; n_b^{(i)}]\!], \ (D_i)_{kk} = \frac{1}{m_{\mathcal{N}_b^{(i)}(k)}} ,$$

- the stiffness scaling:

$$\forall i \in [\![1; N]\!], \forall k \in [\![1; n_b^{(i)}]\!], \ m = \mathcal{N}_b^{(i)}(k) \text{ and } (D_i)_{kk} = \frac{\left(\mathring{A}_{bb}^{(i)} \right)_{kk}}{\sum_{j \in \mathcal{S}_m} \left(\mathring{A}_{bb}^{(j)} \right)_{\mathcal{N}_j^{-1}(m) \mathcal{N}_j^{-1}(m)}} ,$$

- and the coefficient scaling:

$$\forall i \in [\![1;N]\!], \forall k \in [\![1;n_b^{(i)}]\!],\ m = \mathcal{N}_b^{(i)}(k) \text{ and } (D_i)_{kk} = \frac{\rho_m^{(i)}}{\sum\limits_{j \in \mathcal{S}_m} \rho_m^{(j)}},$$

  where ρ_k is a physical coefficient linked to the PDE eq. (1.1), such as a diffusion coefficient or Lamé parameters, associated with the d.o.f. $k \in \mathcal{N}_b$.

From definition 1.11, it appears that the pseudoinverses of the Schur complement can be computed as follows:

$$\forall i \in [\![1;N]\!],\ S_i^\dagger = t^{(i)} \mathring{A}_i^\dagger t^{(i)^T}.$$

Just as in section 1.1.3, a second level will now be added to eq. (1.31). It will be necessary to enforce that the residuals remain in each $\{\text{span}(S_i)\}_{i=1}^N$. For that reason, let $\{R_i\}_{i=1}^N$ be defined as the kernel of each Schur complement. A residual $r \in \mathbb{R}^{\mathcal{N}_b}$ is said to be balanced if:

$$\forall i \in [\![1;N]\!],\ R_i^T D_i B^{(i)^T} r = 0.$$

Let $Z = \begin{bmatrix} B^{(1)} D_1 R_1 & \cdots & B^{(N)} D_N R_N \end{bmatrix}$, then balancing a residual may be achieved using the following projection:

$$P = I - Z \left(Z^T B S B^T Z \right)^{-1} Z^T S. \tag{1.32}$$

The traditional (two-level) BDD preconditioner is then:

$$\begin{aligned} P_{\text{BDD}}^{-1} &= P^T M^{-1} P \\ &= P^T \left(\sum_{i=1}^N B^{(i)} D_i S_i^\dagger D_i B^{(i)^T} \right) P, \end{aligned}$$

using a starting vector u_0 for the iterative method in $\text{span}(Z)$:

$$u_0 = Z \left(Z^T S Z \right)^{-1} Z^T B g.$$

It was shown by Mandel and Brezina [1996], that the condition number $\kappa(P_{\text{BDD}}^{-1} B S B^T)$ is bounded by:

$$\kappa(P_{\text{BDD}}^{-1} B S B^T) < C \left(1 + \log^2 \frac{H}{h} \right), \tag{1.33}$$

where H is the characteristic size of the subdomains, h is the characteristic size of the elements in $\mathcal{T}$, and C is a constant independent of the number of subdomains N, H, h, and the physical coefficients of the underlying PDE.
In recent years, a new variant of the BDD method, called BDDC for Balancing Domain Decomposition by Constraints, has been developed, see for example [Dohrmann 2003; Mandel and Dohrmann 2003].

1.2.2 FETI preconditioner

The Finite Element Tearing and Interconnecting method was historically introduced by Farhat and Roux [1991]. Given a unique interface reaction $\lambda \in \mathbb{R}^M$ such that $\lambda_b = \underline{B}^T \lambda$, it is known, thanks to eq. (1.27), that $B\lambda_b = 0$, i.e. eq. (1.26c) is always verified. Using

the same notation as in the previous section, and denoting by $S^\dagger = \text{diag}(S_1^\dagger, \dots, S_N^\dagger)$ and $R = \text{diag}(R_1, \dots, R_N)$, then a solution of eq. (1.26a) can be decomposed as:

$$u = S^\dagger \underbrace{(g + \underline{B}^T\lambda)}_{\in \text{range} S = \ker R^T} + \underbrace{R\alpha}_{\in \ker S} \tag{1.34a}$$

$$\implies \qquad R^T(g + \underline{B}^T\lambda) = 0. \tag{1.34b}$$

Left multiplying eq. (1.34a) by $\underline{B}$ yields:

$$0 = \underline{B}S^\dagger g + \underline{B}S^\dagger \underline{B}^T\lambda + \underline{B}R\alpha,$$

and introducing $Z = \underline{B}R$, eq. (1.34) reads:

$$\begin{bmatrix} \underline{B}S^\dagger\underline{B}^T & Z \\ Z^T & 0 \end{bmatrix} \begin{bmatrix} \lambda \\ \alpha \end{bmatrix} = \begin{bmatrix} -\underline{B}S^\dagger g \\ -R^T g \end{bmatrix}. \tag{1.35}$$

This system can be solved using a projection P onto $\ker(Z^T)$, i.e. $Z^T P = 0$, and writing λ as $\lambda = \lambda_0 + \tilde{\lambda}$ such that $Z^T\lambda_0 = -R^T g$ and $Z^T\tilde{\lambda} = 0$. Let $M^{-1} \in \mathbb{R}^M \times \mathbb{R}^M$ be a symmetric matrix that will play the role of a one-level preconditioner such that $Z^T M^{-1} Z$ is SPD. Then P can be chosen as:

$$P = I - M^{-1}Z\left(Z^T M^{-1} Z\right)^{-1} Z^T. \tag{1.36}$$

The system now reads:

$$\underline{B}S^\dagger\underline{B}^T(\lambda_0 + \tilde{\lambda}) + Z\alpha = -\underline{B}S^\dagger g,$$

and the unknown α can be eliminated by using P^T:

$$P^T\underline{B}S^\dagger\underline{B}^T\tilde{\lambda} = -P^T\left(\underline{B}S^\dagger\underline{B}^T\lambda_0 + \underline{B}S^\dagger g\right). \tag{1.37}$$

The preconditioner M^{-1} can now be used, and because $\tilde{\lambda}$ must remain in $\ker(Z^T)$, the projection P is applied another time. The traditional (two-level) FETI preconditioner is then:

$$P_{\text{FETI}}^{-1} = PM^{-1}P^T$$

using a starting vector λ_0 for the iterative method that satisfies the admissibility constraints $Z^T\lambda_0 = -R^T g$:

$$\lambda_0 = -M^{-1}Z\left(Z^T M^{-1} Z\right)^{-1} R^T g.$$

Once $\tilde{\lambda}$ is known, α can be determined by left multiplying the first line of eq. (1.35) by $Z^T M^{-1}$:

$$Z^T Q S^\dagger \underline{B}^T Z\lambda + Z^T M^{-1} Z\alpha = -Z^T M^{-1}\underline{B}S^\dagger g$$

$$\implies \qquad \alpha = -\left(Z^T M^{-1} Z\right)^{-1} Z^T Q(\underline{B}S^\dagger g - S^\dagger\underline{B}^T Z\lambda).$$

They are usually three different ways to define M^{-1}. They all rely on the introduction of so-called scaled jump operators $\left\{\tilde{\underline{B}}^{(i)}\right\}_{i=1}^N$ defined as:

$$\forall i \in [\![1; N]\!], \ \tilde{\underline{B}}^{(i)} = \underline{B}^{(i)} D_i\,,$$

where $\{D_i\}_{i=1}^N$ is the same partition of unity as for the BDD preconditioner, cf. page 30. Then, one can define:

- the Dirichlet preconditioner:
$$Q_D = \underline{\tilde{B}}\, S \underline{\tilde{B}}^T,$$
- the lumped preconditioner, where each Schur complement is approximated by canceling the effect of interior d.o.f. on the interface, $\forall j \in [\![1;N]\!]$, $S_j \approx S_j^L = \mathring{A}_{bb}^{(j)} - \cancel{\mathring{A}_{bi}^{(j)} \mathring{A}_{ii}^{(j)-1} \mathring{A}_{ib}^{(j)}}$:
$$Q_L = \underline{\tilde{B}}\, \mathrm{diag}\left(S_1^L, \ldots, S_N^L\right) \underline{\tilde{B}}^T,$$
- the superlumped preconditioner, where the effect of boundary d.o.f. on each other is also canceled, $\forall j \in [\![1;N]\!]$, $S_j \approx S_j^{SL} = \text{diagonal of } \left(\mathring{A}_{bb}^{(j)}\right)$
$$Q_{SL} = \underline{\tilde{B}}\, \mathrm{diag}\left(S_1^{SL}, \ldots, S_N^{SL}\right) \underline{\tilde{B}}^T.$$

Klawonn and Widlund [2001] proved that using $M^{-1} = Q_D$ as a preconditioner, the condition number $\kappa(P_{\text{FETI}}^{-1} \underline{B} S^\dagger \underline{B}^T)$ is bounded by:
$$\kappa(P_{\text{FETI}}^{-1} \underline{B} S^\dagger \underline{B}^T) < C \left(1 + \log \frac{H}{h}\right)^2,$$
using the same notations as for the condition number eq. (1.33). This result was first conjectured by Farhat, Mandel, and Roux [1994]. Restricted to the set of constraints of a single subdomain $i \in [\![1;N]\!]$, the action of the unpreconditioned operator $\underline{B} S^\dagger \underline{B}^T$ on a vector λ yields:
$$\begin{aligned} \underline{B}^{(i)^T} \underline{B} S^\dagger \underline{B}^T \lambda &= \underline{B}^{(i)^T} \sum_{j=1}^{N} \underline{B}^{(j)} S_j^\dagger \underline{B}^{(j)^T} \lambda \\ &= \sum_{j \in \overline{\mathcal{O}}_i} \underline{B}^{(i)^T} \underline{B}^{(j)} S_j^\dagger \lambda_b^{(j)}. \end{aligned} \tag{1.38}$$

In eq. (1.38), there are only local operations, except the matrix-matrix products $\left\{\underline{B}^{(i)^T} \underline{B}^{(j)}\right\}_{\substack{1 \leqslant i \leqslant N \\ j \in \overline{\mathcal{O}}_i}}$. These operators are used to exchange information on the interface between subdomains, and as a consequence, for $i \in [\![1;N]\!]$ and $j \in \overline{\mathcal{O}}_i$, the action of $\underline{B}^{(i)^T} \underline{B}^{(j)}$ on a vector $\lambda^{(j)} \in \mathbb{R}^{n_b^{(j)}}$ can be computed as:
$$\underline{B}^{(i)^T} \underline{B}^{(j)} \lambda^{(j)} = - \sum_{\substack{k \in \mathcal{N}_b^{(i)} \cap \mathcal{N}_b^{(j)} \\ \text{if } \Lambda(i,j,k) \neq 0}} \lambda^{(j)}_{\mathcal{N}_j^{-1}(k)} \varepsilon_{\mathcal{N}_i^{-1}(k)} \in \mathbb{R}^{n_b^{(i)}}. \tag{1.39}$$

A satisfying property of eq. (1.39) is that it does not involve any global information: there is no need for a numbering of all the M constraints. In recent years, other variants of the FETI method have been developed, see for example [Farhat, Lesoinne, Le Tallec, et al. 2001; Farhat, Lesoinne, and Pierson 2000; Klawonn and Rheinbach 2007].

1.3 Improving preconditioners using spectral information: GenEO[4]

In the previous sections, various two-level preconditioners have been presented both in the context of overlapping and nonoverlapping domain decomposition methods. They are all

[4] GenEO stands for Generalized Eigenvalue problem on the Overlap.

based on the introduction of a suitable deflation subspace matrix Z, cf. eqs. (1.16), (1.32) and (1.36). Building such subspace matrices is challenging but they are the backbone of any multigrid or domain decomposition scalable and robust preconditioner. It is well established that in the field of domain decomposition methods, Poincaré–Steklov operators, also known as Dirichlet-to-Neumann operators, play a crucial role, see for example [Bourgat et al. 1988; De Roeck and Le Tallec 1991; Nataf, Rogier, and de Sturler 1994]. For that reason, a lot of effort have been put into approximating them numerically to enhance basic preconditioners as done in for example [Dolean, Nataf, et al. 2012; Magoulès, Roux, and Series 2006; Nataf, Xiang, et al. 2011]. While these methods are efficient and can yield satisfying large-scale numerical results, cf. [Jolivet, Dolean, et al. 2012], some of the previous works were recently extended in [Spillane, Dolean, et al. 2013; Spillane and Rixen 2013]. They propose a way to design preconditioners that are robust with respect to the number of subdomains, the heterogeneities in the PDE, and that are algebraic enough to be applied in both scalar or vectorial spaces, when solving for example system of PDEs. The matrix Z is built using eigenvectors of generalized eigenvalue problems—such as eq. (2.1)—that slow down the convergence of iterative methods, so that it is possible in theory to assemble a preconditioner yielding any theoretical convergence rate chosen a priori. The use of spectral information to enhance preconditioners is not specific to domain decomposition methods, for example in [Chartier et al. 2003] it is done in the context of multigrid methods. In the following paragraphs, it is assumed that a threshold criteria, given next, is used to select $\{\gamma_i\}_{i=1}^N$ eigenpairs per subdomain.

1.3.1 Overlapping methods

The main result of [Spillane, Dolean, et al. 2013] will now be recalled. The same notations as in section 1.1 will be used, and additionally, so are the notions of multiplicity of a d.o.f. given in definition 1.6 and of unassembled operators given in definition 1.7.

Definition 1.12 (restriction on the overlap). *The set of matrices $\left\{\mathring{D}_i\right\}_{i=1}^N$ is defined such that entries of each matrix are nonzero only when associated with d.o.f. on the overlap with their neighboring subdomains:*

$$\forall i \in [\![1;N]\!],\ \left(\mathring{D}_i\right)_{\substack{jk \\ 1\leqslant j\leqslant n_i \\ 1\leqslant k\leqslant n_i}} = \begin{cases} (D_i)_{jk} & \text{if } m_{\mathcal{N}_i(j)} > 1 \\ 0 & \text{otherwise.} \end{cases}$$

These are singular diagonal matrices.

The local generalized eigenvalue problems that will be used from now on are, $\forall i \in [\![1;N]\!]$:

$$\text{find the eigenpairs } \{\lambda_j^{(i)}, v_j^{(i)}\}_{j=1}^{\gamma_i} \text{ such that } \forall j \in [\![1;\gamma_i]\!],\ \mathring{A}_i v_j^{(i)} = \lambda_j^{(i)} \mathring{D}_i \mathring{A}_i \mathring{D}_i v_j^{(i)}. \quad (1.40)$$

Since these sparse problems are purely local, they can be solved concurrently on each subdomain, for example by using ARPACK. After obtaining the eigenpairs, it is possible to build the following deflation subspace matrix Z:

$$Z = \begin{bmatrix} R_1^T W_1 & \cdots & R_N^T W_N \end{bmatrix} \in \mathbb{R}^n \times \mathbb{R}^{\sum_{i=1}^N \gamma_i}, \quad (1.41)$$

where all $\{W_i\}_{i=1}^N$ are dense rectangular matrices in $\mathbb{R}^{n_i} \times \mathbb{R}^{\gamma_i}$ defined as the column-wise concatenation of all $\left\{v_j^{(i)}\right\}_{j=1}^{\gamma_i}$ weighted by the local partition of unity D_i:

$$\forall i \in [\![1;N]\!],\ W_i = \begin{bmatrix} D_i v_1^{(i)} & \cdots & D_i v_{\gamma_i}^{(i)} \end{bmatrix}. \quad (1.42)$$

Proposition 1.10. *If each* $\{\gamma_i\}_{i=1}^N$ *is chosen such that:*

$$\forall i \in [\![1;N]\!],\ \gamma_i = \min\left(j \in [\![1;n_i]\!]\ :\ \lambda_j^{(i)} > \frac{\delta_i}{H_i}\right),$$

and if $m = \max_{j\in[\![1;n]\!]}(m_j)$*, then the following estimate holds for the condition number* $\kappa(P_{AD}^{-1}A)$*:*

$$\kappa(P_{AD}^{-1}A) < (1+m)\left[2 + (2m^2+m)\max_{i\in[\![1;n]\!]}\left(1+\frac{H_i}{\delta_i}\right)\right].$$

This condition number is independent of the number of subdomains and the discretization technique, and is robust with respect to the coefficients of the PDE.

1.3.2 Substructuring methods

The main results of [Dolean, Jolivet, and Nataf 2014] (resp. [Spillane and Rixen 2013]) will now be recalled for the BDD (resp. FETI) preconditioner. In the case of BDD, the generalized eigenvalue problems that are used are, $\forall i \in [\![1;N]\!]$:

$$\text{find the eigenpairs } \{\lambda_j^{(i)}, v_j^{(i)}\}_{j=1}^{\gamma_i} \text{ such that } \forall j \in [\![1;\gamma_i]\!],\ S_i v_j^{(i)} = \lambda_j^{(i)} D_i B^{(i)^T} B S B^T B^{(i)} D_i v_j^{(i)}. \tag{1.43}$$

After obtaining the eigenpairs, it is possible to build the following deflation subspace matrix Z:

$$Z = \begin{bmatrix} B^{(1)}W_1 & \cdots & B^{(N)}W_N \end{bmatrix} \in \mathbb{R}^{\#\mathcal{N}_b} \times \mathbb{R}^{\sum_{i=1}^N \gamma_i}, \tag{1.44}$$

where all $\{W_i\}_{i=1}^N$ are dense rectangular matrices in $\mathbb{R}^{n_b^{(i)}} \times \mathbb{R}^{\gamma_i}$ defined as the column-wise concatenation of all $\left\{v_j^{(i)}\right\}_{j=1}^{\gamma_i}$, cf. eq. (1.42). Note that if $i \in [\![1;N]\!]$ is the index of a floating subdomain, then since S_i is semidefinite, eigenvectors associated with zero eigenvalues span $\ker(S_i)$. In other words, setting $\gamma_i = \dim(\ker(S_i))$, $\forall i \in [\![1;N]\!]$ will yield the original two-level BDD preconditioner, cf. eq. (1.32).

Proposition 1.11. *If* $m = \max_{j\in[\![1;n]\!]}(m_j)$*, then the following estimate holds for the condition number* $\kappa(P_{BDD}^{-1}BSB^T)$*:*

$$\kappa(P_{BDD}^{-1}BSB^T) < \max\left(1, m\max_{i\in[\![1;n]\!]}\left(\frac{1}{\gamma_i}\right)\right). \tag{1.45}$$

In case of FETI, the generalized eigenvalue problems that are used are, $\forall i \in [\![1;N]\!]$:

$$\text{find the eigenpairs } \{\lambda_j^{(i)}, v_j^{(i)}\}_{j=1}^{\gamma_i} \text{ such that } \forall j \in [\![1;\gamma_i]\!],\ S_i v_j^{(i)} = \lambda_j^{(i)} \underline{B}^{(i)^T} M^{-1} \underline{B}^{(i)} v_j^{(i)}. \tag{1.46}$$

After obtaining the eigenpairs, the authors of the method propose to build the following deflation subspace matrix Z:

$$Z = \begin{bmatrix} M^{-1}\underline{B}^{(1)}W_1 & \cdots & M^{-1}\underline{B}^{(N)}W_N \end{bmatrix} \in \mathbb{R}^{M} \times \mathbb{R}^{\sum_{i=1}^N \gamma_i}, \tag{1.47}$$

where all $\{W_i\}_{i=1}^N$ are dense rectangular matrices in $\mathbb{R}^{n_b^{(i)}} \times \mathbb{R}^{\gamma_i}$ defined as the column-wise concatenation of all $\left\{v_j^{(i)}\right\}_{j=1}^{\gamma_i}$, cf. eq. (1.42). While an equivalent estimate as for the BDD preconditioner eq. (1.45) can be proved, the sparsity pattern of the Galerkin operator assembled using the deflation matrix eq. (1.47) is much denser. Hence, this approach is not viable for large-scale problems and the BDD preconditioner will be used instead chapter 5.

Chapter 2

A unified framework

A scientific library to perform domain decomposition methods is now presented. Choices about the design of the library are explained in section 2.1, and the actual implementation is described in section 2.2. Eventually, a simple example for solving a boundary value problem using the finite difference method is given in section 2.3. Readers only interested in numerical experiments should skip to chapter 5.

Une librairie de calcul scientifique pour manier des méthodes de décomposition de domaine est maintenant présentée. Les choix concernant son architecture sont expliqués section 2.1, et l'implémentation est détaillée section 2.2. Pour finir, un exemple simple pour résoudre un problème aux limites en utilisant la méthode des différences finies est donné section 2.3. Le lecteur uniquement intéressé par les résultats numériques est invité à se rendre chapitre 5.

Contents

2.1 Software design 38
2.1.1 Parallel programming model 38
2.1.2 Basic linear algebra 38
2.1.3 Linear solvers 40
2.1.4 Eigenvalue solvers 41
2.1.5 Graph partitioners 42
2.1.6 Similar libraries 42
2.2 Necessary objects 43
2.2.1 Subdomain 43
2.2.2 Coarse operator 46
2.2.3 Sparse and dense eigensolvers for GenEO 50
2.2.4 Iterative methods 51
2.3 A simple prototype 54
2.3.1 Decomposition and partition of unity 54
2.3.2 Finite difference matrices and deflation vectors 56
2.3.3 Instantiation of the preconditioner 57

2.1 Software design

The purpose of this section is to give a broad overview of current possibilities for designing high-performance scientific code. In section 2.1.1, the different models of parallelism appropriate for domain decomposition methods are explored. Sections 2.1.2 to 2.1.5 serve as a comprehensive introduction to necessary numerical tools needed by domain decomposition preconditioners and in section 2.1.6, some frameworks that perform somehow similar tasks are presented.

2.1.1 Parallel programming model

Regarding process interaction, the choice of a model is quite straightforward for implementing efficient parallel domain decomposition methods. To ensure efficient data locality, it is best to distribute each subdomain to different processes that will interact through message passing, with systems like MPI [Snir et al. 1995]. If needed, it is possible to use a finer granularity of process interaction through shared memory multiprocessing programming. This is particularly useful for speeding up computations related to linear algebra that are local to each subdomain, for example the solution of local eigenvalue problems (1.40) and (1.43). It is usually achieved using Pthreads or OpenMP. Both message passing and shared memory models are available within C++, which will be the language used for the library designed in this thesis.

2.1.2 Basic linear algebra

In this section, the focus is on basic linear algebra operations such as matrix-vector products. The first paragraph is related to dense linear algebra while the second is related to sparse linear algebra.

BLAS

Basic Linear Algebra Subprograms are a set of low-level routines that deals with densely stored vectors and matrices [Blackford et al. 2002]. The framework presented in this thesis makes extensive use of the following routines[1]:

1. `?dot`: computes a vector-vector dot product $x^T y$.
2. `?scal`: computes a scalar-vector product $y = \alpha y$.
3. `?axpy`: computes a scalar-vector product and adds the result to a vector $y = \alpha x + y$.
4. `?axpby`: computes two scalar-vector products $y = \alpha x + \beta y$.
5. `?gemv`: computes a scalar-matrix-vector product $y = \alpha Ax + \beta y$.
6. `?symv`: computes a symmetric scalar-matrix-vector product $y = \alpha Ax + \beta y$.
7. `?gemm`: computes a scalar-matrix-matrix product $C = \alpha AB + \beta C$.
8. `?symm`: computes a symmetric scalar-matrix-matrix product $C = \alpha AB + \beta C$.

[1] The question mark in front of each name corresponds to different character codes indicating the data type of the input and output of the routines: `s` (resp. `c`) indicates single precision while `d` (resp. `z`) indicates double precision real (resp. complex) numbers.

The difference between item 5 and item 6 (resp. 7 and 8) is that, since in the latter case A is symmetric, only its upper or lower triangular part is accessed. Items 1 to 6 are memory bound functions, while items 7 and 8 are compute bound functions.

Sparse linear algebra

Intel *Math Kernel Library* is a library which includes an implementation of BLAS, but also routines for sparse linear algebra, statistics, Fast Fourier Transform (FFT)... It is much more complete than the specification in [Duff, Heroux, and Pozo 2002] but unfortunately only runs on Intel-compatible processors. The NVIDIA *CUDA Sparse Matrix library* (cuSPARSE) provides a collection of basic linear algebra subroutines used for sparse matrices that have a similar Application Programming Interface (API) to the MKL. It is used to offload computations on Graphics Processing Units (GPUs). Below is a short description of three common ways to store sparse matrices that can be interpreted by both the MKL and cuSPARSE as well as the other third-party libraries that will be presented in section 2.1.3.
Let A be the following sparse matrix where only its nonzero entries (nnz) are displayed:

$$A = \begin{bmatrix} 1 & 2 & & \\ & 3 & 9 & \\ 2 & & & 1 \\ & 1 & 4 & \end{bmatrix} \qquad \text{nnz}(A) = 8\,.$$

Then A can be represented in:

- COOrdinate format using a triplet (row, col, val) such that all three items are arrays of size $\text{nnz}(A)$ and

$$\forall i \in [\![1; \text{nnz}(A)]\!],\ A_{row[i]\,col[i]} = val[i]\,.$$

$row = [1, 1, 2, 2, 3, 3, 4, 4]$ $\quad$ $col = [1, 2, 2, 3, 1, 4, 2, 3]$ $\quad$ $val = [1, 2, 3, 9, 2, 1, 1, 4]$

- Compressed Sparse Row (CSR) format using a triplet (row_ptr, col, val) such that:
  - col (resp. val) is an array of size $\text{nnz}(A)$ that stores at entry j the column index (resp. value) of nonzero entry j.
  - row_ptr is an array of size $n + 1$, where n is the number of rows of A, that stores at entry j the index of the element in val that is the first nonzero in row j. $row\_ptr[n + 1]$ is set to $\text{nnz}(A) + 1$.

$row\_ptr = [1, 3, 5, 7, 9]$ $\quad$ $col = [1, 2, 2, 3, 1, 4, 2, 3]$ $\quad$ $val = [1, 2, 3, 9, 2, 1, 1, 4]$

- Compressed Sparse Column (CSC) format using a triplet (col_ptr, row, val) such that:
  - row (resp. val) is an array of size $\text{nnz}(A)$ that stores at entry j the row index (resp. value) of nonzero entry j.
  - col_ptr is an array of size $m + 1$, where m is the number of columns of A, that stores at entry j the index of the element in val that is the first nonzero in column j. $col\_ptr[m + 1]$ is set to $\text{nnz}(A) + 1$.

$col\_ptr = [1, 3, 6, 8, 9]$ $\quad$ $row = [1, 3, 1, 2, 3, 2, 4, 4]$ $\quad$ $val = [1, 2, 2, 3, 1, 9, 4, 1]$

If A were to be symmetric, then its CSR and CSC representations would be identical. In that case though, most sparse linear algebra libraries use only either the upper or the lower part of the representation. Note that the upper part of a symmetric CSR representation is identical to the lower part of a CSC representation and vice versa.

The framework presented in this thesis makes extensive use of the following routines[2]:

1. `?csrmv`: computes a scalar-sparse matrix-vector product $y = \alpha Ax + \beta y$.
2. `?csrsymv`: computes a symmetric scalar-sparse matrix-vector product $y = \alpha Ax + \beta y$.
3. `?csrmm`: computes a scalar-sparse matrix-matrix product $Y = \alpha AX + \beta Y$.

2.1.3 Linear solvers

In this section, four state of the art direct solvers used during this thesis will be presented. Most of them are compared by Gould, Scott, and Y. Hu [2007] where they are used to solve symmetric linear systems of equations.

PARDISO

PARDISO is a direct solver based on a supernodal factorization algorithm for solving large sparse symmetric and nonsymmetric linear systems on shared memory and distributed memory architectures written in Fortran [Schenk and Gärtner 2004, 2006; Schenk, Gärtner, and Fichtner 2000]. It can also be used to compute the Schur complement of a matrix. Intel forked the original project to include one of the first version of the software into its own MKL. The two versions have evolved in different directions ever since, but they both use the CSR format as input of the matrix.

MUMPS

MUMPS is a MUltifrontal Massively Parallel sparse direct Solver [Amestoy, Duff, et al. 2001; Amestoy, Guermouche, et al. 2006] written in Fortran. Its main features include the solution of the transposed system, input of the matrix in COO format, error analysis, iterative refinement, out-of-core capability, detection of null pivots, basic estimate of rank deficiency and null space basis for symmetric matrices, and computation of the Schur complement.

PaStiX

PaStiX (Parallel Sparse matriX package) is a parallelized and multithreaded direct solver written in C that provides also an adaptive blockwise iLU(k) factorization that can be used as a parallel preconditioner with approximated supernodes to build a coarser block structure of the incomplete factors [Hénon, Ramet, and Roman 2002]. It can compute the Schur complement of a matrix and uses the CSR format as input of the matrix.

WSMP

The Watson Sparse Matrix Package [Gupta 2000; Gupta and Avron 2000] is a shared and distributed memory parallel linear solver. It is developed at IBM Thomas J. Watson Research Center.

[2] See footnote 1 page 38.

2.1.4 Eigenvalue solvers

In this section, two state of the art eigenvalue solvers used during this thesis will be presented. More precisely, two libraries to solve the following generalized eigenvalue problem will be presented:

$$\text{find the eigenpairs } \{\lambda_i, v_i\}_{i=1}^m \text{ such that } \forall i \in [\![1; m]\!],\ Av_i = \lambda_i B v_i\,. \tag{2.1}$$

The first one will be useful when matrices A and B are sparse matrices, the second one when both are dense matrices. In any case, the matrix B will be invertible so that there is no infinite eigenvalues. For more on the theory of eigenvalue problems, the interested reader is referred to [Parlett 1998].

ARPACK

The ARnoldi PACKage [Lehoucq, Sorensen, and C. Yang 1998] is based on the Implicitly Restarted Arnoldi Method (IRAM) [Arnoldi 1951] and is written in Fortran. One of its most important features is the Reverse Communication Interface (RCI) so that an end-user is free to choose any convenient data structure for the representation of matrices. Instead of solving eq. (2.1), ARPACK solves the following standard eigenvalue problem:

$$\text{find the eigenpairs } \{\nu_i, z_i\}_{i=1}^m \text{ such that } \forall i \in [\![1; m]\!],\ B^{-1}Az_i = \nu_i z_i\,.$$

The user must then provide to the RCI a way for ARPACK to apply B^{-1} to a vector. This can be achieved by using either a sparse direct solver, cf. section 2.1.3, or an iterative method.

LAPACK

Linear Algebra PACKage is a set of routines that deals with densely stored vectors and matrices for solving linear systems, linear least squares, eigenvalue problems and singular value decomposition [Anderson et al. 1999]. For conciseness, the following assumption will be made concerning eq. (2.1): A is symmetric and B is symmetric positive definite (SPD). Hence, it is possible to compute the Cholesky decomposition of B, that is find the unique lower triangular matrix L such that $B = LL^T$. Equation (2.1) is then reduced to the following standard eigenvalue problem:

$$\text{find the eigenpairs } \{\nu_i, z_i\}_{i=1}^m \text{ such that } \forall i \in [\![1; m]\!],\ L^{-1}AL^{T^{-1}}z_i = \nu_i z_i\,. \tag{2.2}$$

The following equality holds for finding the eigenvectors of the generalized eigenvalue problem once problem (2.2) is solved:

$$\forall i \in [\![1; m]\!],\ z_i = L^{-1}y_i\,.$$

The traditional workflow for solving a generalized eigenvalue problem is then to call the following routines[3] (in ascending order):

1. `?potrf`: computes the Cholesky factorization of a SPD matrix.
2. `?sygst`: reduces a symmetric definite generalized eigenvalue problem to a standard form.
3. `?sytrd`: reduces a real symmetric matrix to tridiagonal form.

[3] See footnote 1 page 38.

4. `?stebz`: computes selected eigenvalues of a real symmetric tridiagonal matrix by bisection.
5. `?stein`: computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix..
6. `?ormtr`: transforms the eigenvectors of the standard problem to the ones of the generalized problem.
7. `?trtrs`: solves a system of linear equations with a triangular matrix.

2.1.5 Graph partitioners

In this section, two state of the art graph partitioners used during this thesis will be presented.

METIS

METIS, METIS-MT, and ParMETIS is a set of serial, multithreaded, and parallel programs written in C for partitioning graphs or meshes, and producing fill reducing orderings for sparse matrices. The algorithms implemented in METIS are based on the multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning schemes [Karypis and Kumar 1998].

SCOTCH

SCOTCH and PT-SCOTCH are libraries written in C for sequential and parallel graph partitioning, static mapping and clustering, sequential mesh and hypergraph partitioning, and sequential and parallel sparse matrix block ordering [Chevalier and Pellegrini 2008].

2.1.6 Similar libraries

There are mainly three publicly available libraries that provide functionalities that can be used in the context of domain decomposition.They all have Fortran, C, and C++ interfaces, however they require the assembly of the global system eq. (1.3) or eq. (1.26).

PETSc

The Portable, Extensible Toolkit for Scientific Computation [Balay, Adams, et al. 2014] is a suite of data structures and routines developed in C by Argonne National Laboratory for the parallel solution of scientific applications, modeled by partial differential equations. It is used by many other packages, ranging from eigenvalue problem solvers [Campos et al. 2013] to finite element libraries like Feel++ [Prud'homme et al. 2012], deal.II [Bangerth, Hartmann, and Kanschat 2007], libMesh [Kirk et al. 2006], and FEniCS [Logg, Mardal, and Wells 2012], and has also available interfaces to solvers such as SuperLU [Li 2005] and hypre [Falgout and U. M. Yang 2002]. PETSc is also used as the algebraic backend of SLEPc [Campos et al. 2013], a software library for the solution of large-scale sparse eigenvalue problems, which has an interface to ARPACK, cf. section 2.1.4.

Trilinos

Trilinos [Heroux et al. 2005] is a collection of open source software libraries, called packages, intended to be used as building blocks for the development of scientific applications. It is written in C++ by Sandia National Laboratories and includes multilevel preconditioners like ML [Gee et al. 2006].

DUNE

The Distributed and Unified Numerics Environment [Bastian, Blatt, Dedner, Engwer, Klöfkorn, Kornhuber, et al. 2008; Bastian, Blatt, Dedner, Engwer, Klöfkorn, Ohlberger, et al. 2008] is a modular toolbox for solving partial differential equations with grid-based methods. It supports the easy implementation of the finite element method, the finite volume method, and also the finite difference method. The framework consists of a number of modules which are downloadable as separate packages.

2.2 Necessary objects

All the algorithmic tools needed for performing domain decomposition methods in an object-oriented context will now be described.

2.2.1 Subdomain

The most basic class will be used to store information regarding a subdomain, with no particular data about the underlying preconditioner. For this purpose, a pointer to a sparse matrix is used to access either the local matrix $\{A_{ii}\}_{i=1}^{N}$ eq. (1.11) when using overlapping methods or $\left\{\mathring{A}_i\right\}_{i=1}^{N}$ eq. (1.20) when using nonoverlapping methods, on each process.

Communication buffers

As shown eqs. (1.12), (1.29) and (1.38), communications are needed when simply applying the global operator (respectively A, BSB^T, and $\underline{B}S^{\dagger}\underline{B}^T$) on a global vector (respectively in $\mathbb{R}^n$, $\mathbb{R}^{\#\mathcal{N}_b}$, and $\mathbb{R}^M$), whether a preconditioner is used or not. As a consequence it is best to take care of the necessary components to exchange data between subdomains in the current class. The first obvious step is the creation of buffers that will be needed by MPI each time such a global operation is performed (it will be shown afterwards that they can also be used when applying preconditioners). In practice, on each process $i \in [\![1;N]\!]$, two vectors of $\mathcal{O}_i$ pointers are stored. One is used for the receive buffers, `rb`, the other for the send buffers, `sb`. For all $j \in \mathcal{O}_i$, $\texttt{sb}\left[\mathcal{O}_i^{-1}(j)\right]$ and $\texttt{rb}\left[\mathcal{O}_i^{-1}(j)\right]$ are allocated so that they can store up to $\#\mathcal{N}_i \cap \mathcal{N}_j$ values. An additional object has to be created locally to map local unknowns into the communication buffers. On each process $i \in [\![1;N]\!]$, a vector `mapping` of $\mathcal{O}_i$ pairs of indices and vector of indices is allocated such that:

$$\begin{aligned} \forall j \in [\![1;\#\mathcal{O}_i]\!], \quad & \texttt{mapping}[j].first = \mathcal{O}_i(j) \\ & \forall k \in [\![1;\#\mathcal{N}_i \cap \mathcal{N}_j]\!], \ \texttt{mapping}[j].second[k] = \gamma_{ij}(k)\,, \end{aligned} \tag{2.3}$$

where $\gamma_{ij}(k)$ is the index of the kth d.o.f. of the space Γ_{ij} of $\Omega_i \cap \Omega_j$ in the local space. In general, $\gamma_{ij}(k) \neq \gamma_{ji}(k)$, however, the following equality always holds[4]:

$$\forall k \in [\![1;\#\mathcal{N}_i \cap \mathcal{N}_j]\!], \ \mathcal{N}_i(\gamma_{ij}(k)) = \mathcal{N}_j(\gamma_{ji}(k))\,.$$

[4] As a reminder, for $i \in [\![1;N]\!]$ and $k \in [\![1;n_i]\!]$, $\mathcal{N}_i(k)$ represents what would have been the global index of the kth d.o.f. of the finite element space defined on Ω_i.

Thus, for overlapping methods, the product Au in eq. (1.12) can be computed using the following steps:

- first perform some local operations,

```
out = A_ii D_i in;
```

Algorithm 2.1: Local operations for applying the global operator in overlapping methods.

- then exchange data through the buffers previously allocated.

```
for(int i = 0; i < mapping.size(); ++i) {
   MPI_Irecv(rb[i], mapping[i].second.size(), MPI_DOUBLE, mapping[i].first, tag, comm,
        rq + i);
   for(int j = 0; j < mapping[i].second.size(); ++j)
      sb[i][j] = out[mapping[i].second[j]];
   MPI_Isend(sb[i], mapping[i].second.size(), MPI_DOUBLE, mapping[i].first, tag, comm,
        rq + mapping.size() + i);
}
for(int i = 0; i < mapping.size(); ++i) {
   int index;
   MPI_Waitany(mapping.size(), rq, &index, MPI_STATUS_IGNORE);
   for(int j = 0; j < mapping[index].second.size(); ++j)
      out[mapping[index].second[j]] += rb[index][j];
}
MPI_Waitall(mapping.size(), rq + mapping.size(), MPI_STATUSES_IGNORE);
```

Algorithm 2.2: Communications for applying the global operator in overlapping methods.

For the BDD method, the product $BSB^T u$ in eq. (1.29) can be computed almost exactly as before by just changing the first step, algorithm 2.2 is exactly the same for the communications.

```
out = S_i in;
```

Algorithm 2.3: Local operations for applying the global BDD operator.

For the FETI method, the product $\underline{B}S^\dagger\underline{B}^T\lambda$ in eq. (1.38) is computed slightly differently since it involves signed operators so both steps are modified.

```
u = 0
int i;
MPI_Comm_rank(comm, &i);
for(int j = 0; j < mapping.size(); ++j) {
   if(mapping[j].first < i) {
      for(int k = 0; k < mapping[j].second.size(); ++k)
         u[mapping[j].second[k]] -= in[i][j];
   }
   else {
      for(int k = 0; k < mapping[j].second.size(); ++k)
         u[mapping[j].second[k]] += in[i][j];
   }
}
out = S_i^† u;
```

Algorithm 2.4: Local operations for applying the global FETI operator.

```
for(int j = 0; j < mapping.size(); ++j) {
   MPI_Irecv(rb[j], mapping[j].second.size(), MPI_DOUBLE, mapping[j].first, tag, comm,
        ↪ rq + j);
   if(mapping[j].first < i) {
      for(int k = 0; k < mapping[j].second.size(); ++k)
         out[j][k] = -out[mapping[j].second[k]];
   }
   else {
      for(int k = 0; k < mapping[j].second.size(); ++k)
         out[j][k] = out[mapping[j].second[k]];
   }
   MPI_Isend(out[j], mapping[j].second.size(), MPI_DOUBLE, mapping[j].first, tag, comm,
        ↪ rq + mapping.size() + j);
}
for(int j = 0; j < mapping.size(); ++j) {
   int index;
   MPI_Waitany(mapping.size(), rq, &index, MPI_STATUS_IGNORE);
   for(int k = 0; k < mapping[index].second.size(); ++k)
      out[index][mapping[index].second[j]] += rb[index][j];
}
MPI_Waitall(mapping.size(), rq + mapping.size(), MPI_STATUSES_IGNORE);
```

Algorithm 2.5: Communications for applying the global FETI operator.

While the three previous operations involve global operators, the reformulation done in chapter 1 was valuable for writing algorithms that only require local numbering and peer-to-peer communications.

Preconditioner

An additional class is needed to store information related to the preconditioner being applied. It will manage the data needed for applying the local preconditioner[5], as well as a possible coarse operator. In particular, the local dense rectangular matrices $\{W_i\}_{i=1}^N$ defined eqs. (1.41), (1.44) and (1.47) are stored in this class if a two-level method is used. From this basic class inherit two classes:

1. one used for overlapping Schwarz methods, cf. section 1.1. It stores supplementary information about each local partition of unity $\{D_i\}_{i=1}^N$ which are stored as diagonal matrices (inside a contiguous block of memory of n_i scalars), and also about how to call a third-party library to perform the numerical factorization of each local solver of the base class $\{A_{ii}\}_{i=1}^N$,
2. another one used for nonoverlapping methods. It stores supplementary information about the local number of constraints imposed page 26. From this class inherit two more classes that will store a second solver:
   (a) one used for primal methods like BDD, cf. section 1.2.1.
   (b) another one for dual methods like FETI, cf. section 1.2.2.

   The main difference between these two classes is that in the case of BDD, the solver of the base class represents a numerical factorization of each $\left\{\mathring{A}_i\right\}_{i=1}^N$ which will be used to apply the local Schur complements given definition 1.11, and the solver of the derived class represents a pseudoinverse $\{S_i^\dagger\}_{i=1}^N$. These two roles are switched in the context of dual methods.

[5] Here, shared memory multiprocessing can be useful for increased scalability, cf. section 4.2.

2.2.2 Coarse operator

All classes needed for performing one-level domain decomposition methods have now been introduced. As emphasized in the previous section, it is possible to maintain a certain level of genericity between overlapping and substructuring methods since they share some common building blocks such as the communication patterns and the creation of the MPI buffers or the factorization of the local solvers. The strategy used to extend the framework for two-level methods will now be explained. Once again, abstraction will play a key role so that the ideas behind the assembly and the use of the coarse operator will be applicable for both overlapping and nonoverlapping preconditioners. These ideas have already been partially published in [Jolivet et al. 2014a].

Assembly

When it comes to overlapping Schwarz methods, an important result has to be given for justifying the assembly of the coarse operator.

Corollary 2.1 (of proposition 1.3). *Let $i \in [\![1; N]\!]$ and $u^{(i)} \in \mathbb{R}^{n_i}$. Then,*

$$\begin{aligned} \forall j \in [\![1; N]\!],\ R_j A R_i^T D_i u^{(i)} &= R_j R_i^T R_i A R_i^T D_i u^{(i)} \\ &= R_j R_i^T A_{ii} D_i u^{(i)}. \end{aligned}$$

Proof. Let $v = AR_i^T D_i u^{(i)}$ for $i \in [\![1; N]\!]$ and $u^{(i)} \in \mathbb{R}^{n_i}$. Using proposition 1.2 after proposition 1.3 yields $R_i^T R_i v = v$, hence $R_j v = R_j R_i^T R_i v$. □

Recalling that the Galerkin operator given definition 1.5 is $E = Z^T A Z$ and using the deflation matrix given eq. (1.41), it is possible to exhibit a block structure of E by extending the notations of section 1.3. First, it is important to notice that E is square matrix of order $\sum_{i=1}^{N} \gamma_i$ where $\{\gamma_i\}_{i=1}^{N}$ are the numbers of deflation vectors computed per subdomain using the generalized eigenvalue problem eq. (1.40). An auxiliary function has to be defined:

$$\begin{aligned} \mathcal{R} : \left[\!\!\left[1; \sum_{k=1}^{N} \gamma_k\right]\!\!\right] &\to [\![1; N]\!] \\ j &\mapsto \max\left\{ i : \sum_{k=1}^{i} \gamma_k < j \right\}. \end{aligned} \tag{2.4}$$

$\mathcal{R}$ is the function that maps each column (or row) index of E to its corresponding subdomain index . If all $\{\gamma_i\}_{i=1}^{N}$ equal γ (e.g. when a uniform criterion is chosen proposition 1.10) then this function can be simplified:

$$\mathcal{R}(i) = \left\lfloor \frac{i}{\gamma} \right\rfloor.$$

The structure of the coarse operator E may now be written using blocks of size $\gamma_{\mathcal{R}(i)} \times \gamma_{\mathcal{R}(j)}$, for all row and column indices $(i, j) \in [\![1; \sum_{k=1}^{N} \gamma_k]\!]^2$:

$$E_{\mathcal{R}(i)\mathcal{R}(j)} = W_{\mathcal{R}(i)}^T R_{\mathcal{R}(i)}^T A R_{\mathcal{R}(j)} W_{\mathcal{R}(j)} \in \mathbb{R}^{\gamma_{\mathcal{R}(i)}} \times \mathbb{R}^{\gamma_{\mathcal{R}(j)}}.$$

Only the block structure will be used from now on unless stated otherwise, therefore the function $\mathcal{R}$ will be dropped, instead, for all block indices $(i, j) \in [\![1; N]\!]^2$:

$$E_{ij} = W_i^T R_i^T A R_j W_j \in \mathbb{R}^{\gamma_i} \times \mathbb{R}^{\gamma_j}.$$

Proposition 2.2. *The sparsity pattern of E is determined by the connectivity of the graph where each subdomain is a vertex in the said graph and there is an edge between each neighboring subdomains. Moreover, thanks to corollary 2.1, the following equality holds $\forall (i,j) \in [\![1;N]\!]^2$:*

$$E_{ij} = W_i^T R_i R_j^T A_{jj} W_j \in \mathbb{R}^{\gamma_i} \times \mathbb{R}^{\gamma_j} . \tag{2.6}$$

Proof. It is a direct consequence of the block structure of E given eq. (2.6) and of proposition 1.1. □

In eq. (2.6), there are only local operations, except the matrix-matrix product $\left\{R_i R_j^T\right\}_{\substack{1 \leqslant i \leqslant N \\ j \in \overline{\mathcal{O}}_i}}$ just as in eq. (1.12). It is really important to note once again that there is no need either of the global system A or of the global deflation matrix Z.

At this time, it is supposed that E is stored using a row-wise distribution. To minimize communication overhead, each process $i \in [\![1;N]\!]$ is in charge of assembling all blocks $\{E_{ij}\}_{j \in \overline{\mathcal{O}}_i}$. This is done concurrently on each subdomain by following a workflow similar to the one described section 2.2.1:

1. first scale the eigenvectors $\left\{v_j^{(i)}\right\}_{j=1}^{\gamma_i}$ by the local partition of unity D_i using `gemv`, cf. section 2.1.2,
2. then compute the local sparse matrix-matrix product $A_{ii} W_i$ using `csrmm`,
3. send to each neighbor $j \in \mathcal{O}_i$ the result restricted to the duplicated unknowns $\mathcal{S}_j^{(i)} = R_j R_i^T A_{ii} W_i$ and receive from each neighbor $\mathcal{R}_j^{(i)} = R_i R_j^T A_{jj} W_j$,
4. compute the diagonal block $E_{ii} = W_i^T A_{ii} W_i$ using `gemm`,
5. compute the off-diagonal block after reception $E_{ij} = W_i^T \mathcal{R}_j^{(i)}$ using `gemm`.

Steps 1 and 2 (resp. 3) are the exact counterparts of algorithm 2.1 (resp. 2.2) when multiple vectors are used as input. Step 3 can be overlapped with step 4.

The assembly of the coarse operator for nonoverlapping preconditioners will now be investigated. If the FETI method is being used, then the Galerkin operator is defined as $E = Z^T M^{-1} Z$. When $M^{-1} = Q_{SL}$, then using the same block structure as for the assembly of the coarse operator in overlapping Schwarz methods, for all block indices $(i,j) \in [\![1;N]\!]^2$:

$$\begin{aligned} E_{ij} &= R_i^T \underline{B}^{(i)^T} \sum_{k=1}^{N} \tilde{\underline{B}}^{(k)} S_k^{SL} \tilde{\underline{B}}^{(k)^T} \underline{B}^{(j)} R_j \\ &= R_i^T \underline{B}^{(i)^T} \sum_{k \in \overline{\mathcal{O}}_j} \tilde{\underline{B}}^{(k)} S_k^{SL} \tilde{\underline{B}}^{(k)^T} \underline{B}^{(j)} R_j \in \mathbb{R}^{\dim(\ker(S_i))} \times \mathbb{R}^{\dim(\ker(S_j))} . \end{aligned}$$

Since the matrices $\left\{S_i^{SL}\right\}_{i=1}^{N}$ are diagonal, the sparsity pattern of E is the same as for overlapping preconditioners, and the following simplification can be made:

$$E_{ij} = R_i^T \underline{B}^{(i)^T} \sum_{k \in \overline{\mathcal{O}}_i \cap \overline{\mathcal{O}}_j} \tilde{\underline{B}}^{(k)} S_k^{SL} \tilde{\underline{B}}^{(k)^T} \underline{B}^{(j)} R_j .$$

When $M^{-1} = Q_D$ or Q_L, the local Schur complement are either not approximated, or approximated by matrices which have nonzero entries outside their main diagonal. Hence, the previous simplification is not possible anymore, instead,

$$E_{ij} = R_i^T \underline{B}^{(i)^T} \sum_{k \in \overline{\mathcal{O}}_i \cup \overline{\mathcal{O}}_j} \underline{\tilde{B}}^{(k)} S_k^\star \underline{\tilde{B}}^{(k)^T} \underline{B}^{(j)} R_j \,.$$

where $S_k^\star = S_k$ or S_k^L. The sparsity pattern of E is then determined by the connectivity of the graph where each subdomain is a vertex in the said graph and there is an edge between each neighboring subdomain and between each neighboring subdomain of each neighboring subdomain. As for overlapping methods, each process $i \in [\![1; N]\!]$ is in charge of assembling all blocks $\{E_{ij}\}_{\substack{j \in \mathcal{O}_k \\ k \in \overline{\mathcal{O}}_i}}$. This is done concurrently on each subdomain by following a workflow similar to the one described section 2.2.1:

1. send to each neighbor $j \in \mathcal{O}_i$ the deflation vectors which could only be the rigid body modes restricted to the common interface $\mathcal{S}_j^{(i)} = B^{(j)^T} B^{(i)} R_i$ and receive from each neighbor $\mathcal{R}_j^{(i)} = B^{(i)^T} B^{(j)} R_j$,

2. append all received vectors to the local ones in a dense matrix $\mathcal{L}^{(i)} = \begin{bmatrix} R_i & \mathcal{R}_j^{(i)} \end{bmatrix}_{j \in \mathcal{O}_i}$ and then scale $\mathcal{L}^{(i)}$ by the local partition of unity D_i using `gemv` and left apply the local preconditioner S_i, S_i^L, or S_i^{SL}. Scale again the result using `gemv`.

For the following items in this list, given a local matrix made of restrictions of deflation vectors $\mathcal{L}^{(i)}$, $\forall j \in [\![1; N]\!]$, $\mathcal{L}^{(i)}[j]$ will refer to the block of the matrix made from vectors of subdomain j. For example, at this point in the algorithm, $\forall i \in [\![1; N]\!]$, $\mathcal{L}^{(i)}[i] = D_i S_i^\star D_i R_i$. Moreover, $\forall i \in [\![1; N]\!]$, let $\mathcal{Q}_i = \mathcal{O}_i \cup \{k \in [\![1; N]\!] \,:\, (\exists j \in \mathcal{O}_i \,:\, k \in \mathcal{O}_j)\}$. Thus,

3. initialize for all $j \in \overline{\mathcal{O}}_i$,
$$\mathcal{V}_j^{(i)} = \mathcal{L}^{(i)}[j] \,,$$
and for all $k \in \mathcal{Q}_j \setminus \overline{\mathcal{O}}_j$,
$$\mathcal{V}_k^{(i)} = 0 \,,$$

4. then send to each neighbor $j \in \mathcal{O}_i$ the matrices made of $\sum_{k \in \overline{\mathcal{O}}_i} \gamma_k$ column vectors $\mathcal{S}_j'^{(i)} = B^{(j)^T} B^{(i)} \mathcal{L}^{(i)}$ and receive from each neighbor $\mathcal{R}_j'^{(i)} = B^{(i)^T} B^{(j)} \mathcal{L}^{(j)}$,

5. and finally, perform the reduction step for all $j \in \mathcal{O}_i$,
$$\forall k \in \overline{\mathcal{O}}_j,\ \mathcal{V}_k^{(i)} \mathrel{+}= \mathcal{R}_j'^{(i)}[k] \,,$$
followed by a single call to `gemm` used to left multiply by R_i^T:
$$\forall j \in \mathcal{Q}_i \cup \{i\},\ E_{ij} = R_i^T \mathcal{V}_j^{(i)} \,.^{6}$$

Steps 1 and 2 are the counterparts of algorithms 2.4 and 2.5 when using scaled jump operators and multiple vectors as input[7]. Step 3 can be overlapped with step 4.

[6] Since E is stored in CSR format and local blocks of rows of E are stored contiguously, there is really only one call to `gemm` even if the column indices j are not contiguous.

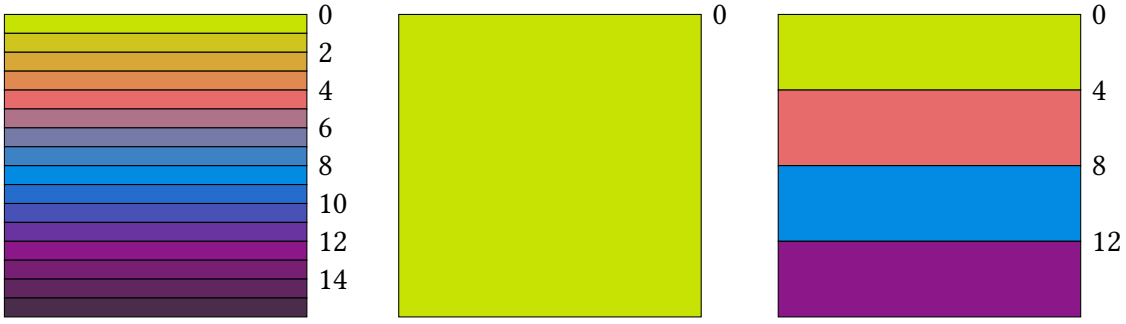
[7] Using modern linear solvers, it is generally much more efficient in terms of memory and FLOP to perform a single forward elimination and back substitution with multiple right-hand sides, than multiple forward eliminations and back substitutions with a single RHS each time.

Action of E^{-1}

Once the Galerkin matrix has been fully assembled, it is only used to solve coarse problems, that is, the action of E^{-1} must be known. As explained in the introductory section ii.1, this may be done using either an iterative method or a direct method. However, since the coarse problem is involved at each application of a two-level preconditioner, it is important to use a robust and fast method. As a consequence, a direct solver is usually preferred to first factorize the sparse matrix E during a preprocessing step, and then to perform forward eliminations and back substitutions each time the action of E^{-1} is needed. Because the number of deflation vectors per subdomain is usually rather low (less than 50), there must be some kind of transformation operating on E so that its distribution is less fine-grained, cf. fig. 2.6. Otherwise, direct solvers will communicate too much when trying to factorize it. Moreover, it might be convenient for the user to choose a number of MPI processes involved in the factorization of E different than the number of subdomains N.

Coarse correction

It is assumed here that the Galerkin matrix has been centralized on a single MPI process that will be referred to as a master process. This approach is ineffective in practice because the size of E might become too large to be factorized in a timely manner by only one process, but that way it is possible to give a simpler insight of the computations and communications induced by a coarse correction. A more sophisticated approach where the number of master processes is greater than one is explained section 4.1 and depicted fig. 2.6.



(a) Finest-grained distribution **(b)** Coarsest-grained distribution **(c)** Efficient distribution

Fig 2.6: Different distributions of the coarse operator E. Each color is associated with a chunk of row of E stored on a process whose rank is displayed on the right—for clarity, in fig. 2.6a, the odd ranks are not displayed.

Once the action of E^{-1} can be computed, the last step is to actually perform a coarse correction, that is, given a global vector (respectively in $\mathbb{R}^n$, $\mathbb{R}^{\#\mathcal{N}_b}$, and $\mathbb{R}^M$), apply on the left the product $ZE^{-1}Z^T$. This is represented fig. 2.7 when an overlapping four-way decomposition is used. Each part of a matrix or vector owned by a subdomain is color-coded: this is a formal representation since in practice, the deflation matrix Z is not assembled and there is no notion of distributed vector like u. A coarse correction can then be broken down into four elementary operations:

1. compute locally $v^{(i)} = W_i^T u^{(i)}$ and gather those values inside v on a master process,
2. compute $x = E^{-1}v$ on a master process,
3. scatter x from a master process inside $x^{(i)}$ on each process and compute locally $y^{(i)} = W_i x^{(i)}$,

4. perform the reduction $u^{(i)} = R_i \sum_{j \in \overline{\mathcal{O}}_i} R_j^T y^{(j)}$ using peer-to-peer communications[8].

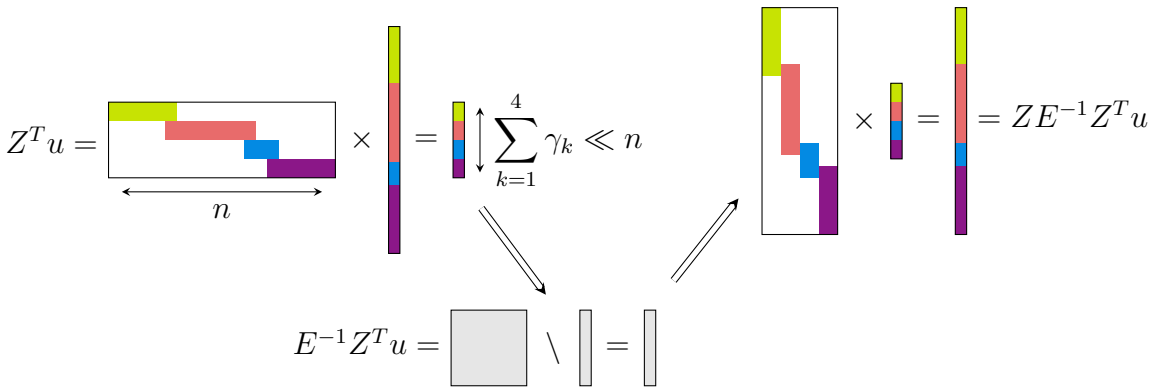


Fig 2.7: Representation of the four algebraic operations performed during one coarse correction.

2.2.3 Sparse and dense eigensolvers for GenEO

In the previous paragraph, the deflation vectors used were not defined explicitly, so that a user could try his/her own coarse operator. The framework provides ways to assemble the GenEO coarse space by solving eq. (1.40) in the case of overlapping methods, or eq. (1.43) in the case of the BDD method.

For overlapping preconditioners, the user has to provide each local unassembled operator $\left\{\mathring{A}_i\right\}_{i=1}^N$ because it is not possible to retrieve these from the local assembled operators $\{A_{ii}\}_{i=1}^N$ algebraically—the discretization technique plays a crucial role here. The right-hand side matrix of eq. (1.40) on subdomain $i \in [\![1; N]\!]$ is assembled by:

1. setting the rows and columns of the unassembled operator to 0 for unknowns $j \in [\![1; n_i]\!]$ that are outside of the overlap, i.e. $m_{\mathcal{N}_i(j)} = 1$,

2. scaling nonzero entries associated with unknowns $(j, k) \in [\![1; n_i]\!]^2$ that are inside of the overlap by the values of the local partition of unity associated with these unknowns: $(D_i)_{jj} \times (D_i)_{kk}$.

The matrix pencils $\left\{(\mathring{A}_i, \mathring{D}_i \mathring{A}_i \mathring{D}_i)\right\}_{i=1}^N$ are then passed over to ARPACK which returns the GenEO deflation vectors that may be used for assembling the Galerkin matrix E.

For nonoverlapping preconditioners, the local generalized eigenvalue problems (1.43) (resp. (1.46)) involve global structures such as the primal assembly operator B and the Schur complement S (resp. the scaled dual jump operator $\underline{\tilde{B}}$ and either the Schur complement or one of its approximation defined page 33). It is once again possible to bypass the global assembly of these operators by taking a closer look at the right-hand side of these problems. For the sake of simplicity, only the case of the primal problem (1.43) will be considered, but the same methodology could be applied to the dual problem eq. (1.46). Naively, the following local operators have to be assembled, $\forall i \in [\![1; N]\!]$:

$$S_i^{\text{local}} = D_i B^{(i)^T} B S B^T B^{(i)} D_i = \sum_{j \in \overline{\mathcal{O}}_i} D_i B^{(i)^T} B^{(j)} S_j B^{(j)^T} B^{(i)} D_i \,.$$

[8] With the BDD method, this becomes $u^{(i)} = B^{(i)} \sum_{j \in \overline{\mathcal{O}}_i} B^{(j)^T} y^{(j)}$, and with the FETI method, this becomes $u^{(i)} = \underline{B}^{(i)} \sum_{j \in \overline{\mathcal{O}}_i} \underline{B}^{(j)^T} y^{(j)}$.

S_i^{local}, $\forall i \in [\![1; N]\!]$, is the scaled restriction of the global Schur complement S to the indices of boundary d.o.f. of subdomain i. This equation is similar to eq. (1.29), and the matrix-matrix products $\left\{B^{(i)^T} B^{(j)}\right\}_{\substack{1 \leqslant i \leqslant N \\ j \in \overline{\mathcal{O}}_i}}$ represent data transfer between subdomains across interfaces. By using a direct solver such as MUMPS or PARDISO as introduced section 2.1.3, it is possible for each structure $i \in [\![1; N]\!]$ to retrieve its local Schur complement S_i. The workflow to assemble S_i^{local} is then uncomplicated:

- initialize $S_i^{\text{local}} = S_i$,
- send to each neighbor $j \in \mathcal{O}_i$ the local Schur complement restricted to the common interface $\mathcal{S}_j^{(i)} = B^{(j)^T} B^{(i)} S_i B^{(i)^T} B^{(j)}$ and receive from each neighbor $\mathcal{R}_j^{(i)} = B^{(i)^T} B^{(j)} S_j B^{(j)^T} B^{(i)}$, these point-to-point messages have sizes that are the square of the message sizes when communicating for applying the BDD operator, cf. algorithm 2.2,
- accumulate the received messages from each neighbor $j \in \mathcal{O}_i$ into S_i^{local}, i.e. $S_i^{\text{local}} \mathrel{+}= \sum_{j \in \mathcal{O}_i} \mathcal{R}_j^{(i)}$,
- scale each entry $\left(S_i^{\text{local}}\right)_{jk}{}_{1 \leqslant j,k \leqslant n_b^{(i)}}$ by $(D_i)_{jj} \times (D_i)_{kk}$.

The matrix pencils $\left\{(S_i, S_i^{\text{local}})\right\}_{i=1}^N$ are then passed over to LAPACK which returns the GenEO deflation vectors that may be used for assembling the Galerkin matrix E.

2.2.4 Iterative methods

Every aspect of the implementation of the methods introduced and reformulated in chapter 1 has now been considered. The final step when it comes down to these preconditioners is to use them inside an iterative method.

Preconditioned Conjugate Gradient

The Conjugate Gradient (CG) introduced by Hestenes and Stiefel [1952] is among the most practical techniques for solving SPD systems of equations. In the context of domain decomposition methods, this will be particularly convenient when using either the BDD or FETI method, which can be embedded inside the CG.

Algorithm 2.8: Preconditioned CG.

Input: uncondensed RHS f and first guess x_0, tolerance ε, number of iterations m

1 $i \leftarrow 0$
2 initialize(x_i, f, y_i, r_i)
3 $z_i \leftarrow M^{-1} r_i$
4 $\varepsilon_i \leftarrow \sqrt{(z_i, z_i)}$
5 **while** $\frac{\varepsilon_i}{\varepsilon_0} < \varepsilon$ **and** $i{+}{+} < m$ **do**
6 $\quad p_i \leftarrow P z_{i-1}$ // P is defined eq. (1.32) for the BDD method
// (resp. (1.36)) (resp. FETI)
7 $\quad$ **for** $k \leftarrow 0$ **to** $i-2$ **do**
8 $\quad\quad \alpha_k \leftarrow (z_k, p_i)$
9 $\quad$ **for** $k \leftarrow 0$ **to** $i-2$ **do**
10 $\quad\quad p_i \leftarrow p_i - \frac{\alpha_k}{\beta_k} p_k$
11 $\quad z_i \leftarrow \mathcal{S} p_i$
// $\mathcal{S} = BSB^T$ (resp. $\underline{B} S^\dagger \underline{B}^T$) for the BDD (resp. FETI) method
12 $\quad \beta_{i-1} \leftarrow (z_{i-1}, z_i)$ $\quad \beta_i \leftarrow (r_{i-1}, z_i)$
13 $\quad y_i \leftarrow \frac{\beta_i}{\beta_{i-1}} p_i$ $\quad r_i \leftarrow -\frac{\beta_i}{\beta_{i-1}} z_i$
14 $\quad r_i \leftarrow P^T r_i$
15 $\quad z_{i+1} \leftarrow M^{-1} r_i$
16 $\quad \varepsilon_i \leftarrow \sqrt{(z_i, z_i)}$

Function initialize(x, f, y, r): // for the BDD method
1 $\quad g \leftarrow B(f_b - \mathring{A}_{bi} \mathring{A}_{ii}^{-1} f_i)$
2 $\quad y \leftarrow Z(Z^T B S B^T Z)^{-1} Z^T g$
3 $\quad r \leftarrow g - Sy$

Function initialize(x, f, y, r): // for the FETI method
1 $\quad y \leftarrow M^{-1} Z (Z^T M^{-1} Z)^{-1} R^T f$
2 $\quad x \leftarrow S^\dagger f$
3 $\quad r \leftarrow P^T \underline{B} (x - S^\dagger \underline{B}^T y)$

Additionally, for computing the final solution x on the interface using the Lagrange multiplier y at the end of the convergence of the CG when using the FETI method, the following auxiliary function is needed.

Function solution(x, y):
$\quad x \leftarrow x - S^\dagger \underline{B}^T y$
$\quad x \leftarrow x - R(Z^T M^{-1} Z)^{-1} Z^T \underline{B}^T M^{-1} \underline{B} x$

To ensure orthogonality between search directions, a classical Gram-Schmidt process is implemented lines 7–10. The scalar products lines 12 and 16 require one global reduction.

Preconditioned Generalized Minimal RESidual method

The Generalized Minimal RESidual method (GMRES) introduced by Saad and Schultz [1986] may be used for solving nonsymmetric systems of equations. This will be valuable when

using the Restricted additive Schwarz method introduced section 1.1.2 because the preconditioner is in this case nonsymmetric, even if the original system is.

Algorithm 2.9: Preconditioned GMRES.

Input: RHS f and first guess x_0, tolerance ε, number of iterations m

$i \leftarrow 0$

$w \leftarrow P^{-1} f$

$\varepsilon_0 \leftarrow \sqrt{(w, w)}$

$y \leftarrow b - Ax_i$

$w \leftarrow P^{-1} y$

$s_{i+1} \leftarrow \sqrt{(w, w)}$

$V_{:\,i+1} \leftarrow \dfrac{w}{s_{i+1}}$

while $i{+}{+} < m$ **and** $\dfrac{s_i}{\varepsilon_0} < \varepsilon$ **do**

- $w \leftarrow P^{-1} A V_{:\,i}$
- **for** $k \leftarrow 1$ **to** i **do**
  - $H_{ik} \leftarrow (V_{:\,k}, w)$
- **for** $k \leftarrow 1$ **to** i **do**
  - $w \leftarrow w - H_{i\,:} V_{:\,k}$
- $H_{i\,i+1} \leftarrow \sqrt{(w, w)}$
- $V_{:\,i+1} \leftarrow \dfrac{w}{H_{i\,i+1}}$
- **for** $k \leftarrow 1$ **to** $i - 1$ **do**
  - $\gamma \leftarrow cs_k H_{ik} + sn_k H_{i\,k+1}$
  - $H_{i\,k+1} \leftarrow -sn_k H_{ik} + cs_k H_{i\,k+1}$
  - $H_{ik} \leftarrow \gamma$
- $\delta \leftarrow \sqrt{H_{ii}^2 + H_{i\,i+1}^2}$
- $cs_i \leftarrow \dfrac{H_{ii}}{\delta} \qquad sn_i \leftarrow \dfrac{H_{i\,i+1}}{\delta}$
- $H_{i\,i+1} \leftarrow cs_i H_{ii} + sn_i H_{i\,i+1}$
- $s_{i+1} \leftarrow -sn_i s_i$
- $s_i \leftarrow s_i cs_i$

solution(x_0, i, s, V, H)

Function solution(x, i, s, V, H)**:**

- **for** $k \leftarrow i$ **to** 1 **do**
  - $s_i \leftarrow \dfrac{s_i}{H_{ii}}$
  - $s \leftarrow s - s_i H_{i\,:}$
- **return** $x + \sum_{j=1}^{i} s_j V_{:\,j}$

In practice, if the maximum number of iterations m is too large, it may be beneficial to use a smaller number p of Krylov directions to orthogonalize against and restart the GMRES every p iterations.

2.3 A simple prototype

The goal of this section is to provide a standalone code built on top of the framework detailed in the previous paragraphs. The problem solved by this code is Poisson's two-dimensional eq. (1.5) on a square Ω, using a five-point stencil finite difference discretization on a uniform grid. In sections 2.3.1 and 2.3.2, the data structures needed for initializing a two-level overlapping Schwarz preconditioner are presented. In section 2.3.3, the framework is instantiated and the linear system is solved. It is assumed that the code is called with four arguments:

- `Nx` and `Ny`, the number of discretization points in each direction,
- `overlap`, the level of overlap,
- `stop`, the relative residual error that has to be reached for the iterative method to stop.

Furthermore, all processes of the default communicator `MPI_COMM_WORLD` are used in the decomposition.

2.3.1 Decomposition and partition of unity

A simple Cartesian decomposition of the square $\Omega = [0; L]^2$ is used. Accordingly, it is possible to get the starting and ending indices of the discretization points in each direction for any subdomain.

```
37 int rankWorld;
   int sizeWorld;
   MPI_Comm_size(MPI_COMM_WORLD, &sizeWorld);
   MPI_Comm_rank(MPI_COMM_WORLD, &rankWorld);
   int xGrid = int(sqrt(sizeWorld));
42 while(sizeWorld % xGrid != 0)
      --xGrid;
   int yGrid = sizeWorld / xGrid;

   int y = rankWorld / xGrid;
47 int x = rankWorld - xGrid * y;

   int iStart = std::max(x * Nx / xGrid - overlap, 0);
   int iEnd = std::min((x + 1) * Nx / xGrid + overlap, Nx);
   int jStart = std::max(y * Ny / yGrid - overlap, 0);
52 int jEnd = std::min((y + 1) * Ny / yGrid + overlap, Ny);
   int ndof = (iEnd - iStart) * (jEnd - jStart);
   int nnz = ndof * 3 - (iEnd - iStart) - (jEnd - jStart);
```

Algorithm 2.10: Decomposition of Ω into regular rectangles.

Then, for each subdomain $i \in [\![1; N]\!]$, it is possible to build the partition of unity D_i, as well as the functions $\{\gamma_{ij}\}_{j \in \mathcal{O}_i}$ introduced eq. (2.3) that will be useful for the communications between subdomains. The approach used to build the partition of unity is similar to the one proposed eq. (1.6): first the functions $\{\widetilde{\chi}_i\}_{i=1}^N$ will be computed, then the framework will be in charge of exchanging the appropriate values to build the final partition of unity. Only the construction of these operators restricted to the intersection with the neighboring subdomains located below in the decomposition is reviewed, the extension to neighboring subdomains elsewhere—above, on the left and right—is undemanding.

```
double* d = new double[ndof]; std::fill(d, d + ndof, 1.0);
std::vector<std::vector<int>*> mapping; mapping.reserve(8);
std::vector<int> o; o.reserve(8); // at most eight neighbors in 2D
if(jStart != 0) { // this subdomain doesn't touch the bottom side of Ω
  if(iStart != 0) { // this subd. doesn't touch the left side of Ω
    o.push_back(rankWorld - xGrid - 1); // subd. on the lower left corner is a neighbor
    mapping.push_back(new std::vector<int>());
    mapping.back()->reserve(4 * overlap * overlap);
    for(int j = 0; j < 2 * overlap; ++j)
      for(int i = iStart; i < iStart + 2 * overlap; ++i)
        mapping.back()->push_back(i - iStart + (iEnd - iStart) * j);
    for(int j = 0; j < overlap; ++j) {
      for(int i = 0; i < overlap - j; ++i)
        d[i + j + j * (iEnd - iStart)] = j / (double)overlap;
      for(int i = 0; i < j; ++i)
        d[i + j * (iEnd - iStart)] = i / (double)overlap;
    }
  }
  else // this subd. touches the left side of Ω
    for(int j = 0; j < overlap; ++j)
      for(int i = 0; i < overlap; ++i)
        d[i + j * (iEnd - iStart)] = j / (double)overlap;
  o.push_back(rankWorld - xGrid); // subd. below is a neighbor
  mapping.push_back(new std::vector<int>());
  mapping.back()->reserve(2 * overlap * (iEnd - iStart));
  for(int j = 0; j < 2 * overlap; ++j)
    for(int i = iStart; i < iEnd; ++i)
      mapping.back()->push_back(i - iStart + (iEnd - iStart) * j);
  for(int j = 0; j < overlap; ++j)
    for(int i = iStart + overlap; i < iEnd - overlap; ++i)
      d[i - iStart + (iEnd - iStart) * j] = j / (double)overlap;
  if(iEnd != Nx) { // this subd. doesn't touch the right side of Ω
    o.push_back(rankWorld - xGrid + 1); // subd. on the lower right corner is a neighbor
    mapping.push_back(new std::vector<int>());
    mapping.back()->reserve(4 * overlap * overlap);
    for(int i = 0; i < 2 * overlap; ++i)
      for(int j = 0; j < 2 * overlap; ++j)
        mapping.back()->push_back((iEnd - iStart) * (i + 1) - 2 * overlap + j);
    for(int j = 0; j < overlap; ++j) {
      for(int i = 0; i < overlap - j; ++i)
        d[(iEnd - iStart) * (j + 1) - overlap + i] = j / (double)overlap;
      for(int i = 0; i < j; ++i)
        d[(iEnd - iStart) * (j + 1) - i - 1] = i / (double)overlap;
    }
  }
  else
    for(int j = 0; j < overlap; ++j)
      for(int i = 0; i < overlap; ++i)
        d[(iEnd - iStart) * (j + 1) - overlap + i] = j / (double)overlap;
}
```

Algorithm 2.11: Construction of the basic structures.

2.3.2 Finite difference matrices and deflation vectors

A standard second order discretization of the Laplace operator in Poisson's eq. (1.5) leads to the following system of linear equations:

$$\forall (i,j) \in [\![1; N_x - 1]\!] \times [\![1; N_y - 1]\!], \quad \frac{-u_{i-1\,j} + 2u_{ij} - u_{i+1\,j}}{\Delta x^2} + \frac{-u_{i\,j-1} + 2u_{ij} - u_{i\,j+1}}{\Delta y^2} = f_{ij}$$
$$\forall (i,j) \in [\![0; N_x]\!] \times [\![0; N_y]\!], \quad u_{i0} = u_{0j} = u_{iN_y} = u_{N_x j} = 0\,,$$

where $u_{ij} \approx u(i\Delta x, j\Delta y)$ and $f_{ij} \approx f(i\Delta x, j\Delta y)$, $\forall (i,j) \in [\![0; N_x]\!] \times [\![0; N_y]\!]$. Ordering unknowns row-by-row, the solution vector looks like:

$$u = \begin{bmatrix} u_{00} & u_{10} & \cdots & u_{N_x 0} & u_{01} & \cdots & u_{N_x-1\,N_y} & u_{N_x N_y} \end{bmatrix}^T,$$

and this leads to a well-known block triangular SPD linear system $Au = f$. From this system, each local matrix $\{A_{ii}\}_{i=1}^{N}$ can be easily constructed concurrently as shown below. Since these matrices are symmetric, only their upper triangular parts are assembled.

```
ia = new int[ndof + 1];
ja = new int[nnz];
a = new double[nnz];
ia[0] = 0;
ia[ndof] = nnz;
for(int j = jStart, k = 0, nnz = 0; j < jEnd; ++j) {
   for(int i = iStart; i < iEnd; ++i) {
      if(j > jStart) { // this d.o.f. is not on the bottom side of the subd.
         a[nnz] = -1 / (dy * dy);
         ja[nnz++] = k - (Ny / yGrid);
      }
      if(i > iStart) { // this d.o.f. is not on the left side of the subd.
         a[nnz] = -1 / (dx * dx);
         ja[nnz++] = k - 1;
      }
      a[nnz] = 2 / (dx * dx) + 2 / (dy * dy);
      ja[nnz++] = k;
      ia[++k] = nnz;
   }
}
```

Algorithm 2.12: Assembly of the local matrices.

The final step to ensure scalability of the solver is to compute deflation vectors that will be used for building a coarse operator as in section 2.2.2. These vectors could be computed by solving the generalized eigenvalue problem eq. (1.40) for example, but since an homogeneous problem is being investigated, the coarse space introduced by Nicolaides [1987] should suffice, cf. eq. (1.18). Note that the deflation vectors will be automatically multiplied by the local partition of unity when assembling the coarse operator, hence the initial vector must be set to 1 at first.

```
double** deflation = new double*[1];
*deflation = new double[ndof];
std::fill(*deflation, *deflation + ndof, 1.0);
```

Algorithm 2.13: Construction of the local deflation vectors.

2.3.3 Instantiation of the preconditioner

Each and every object of the framework described in section 2.2 can now be initialized with the structures computed previously. The first step is to instantiate a preconditioner with a local solver `SUBDOMAIN` and a distributed solver `COARSEOPERATOR` for a potential coarse operator. Each local solver can be different between subdomains, but the distributed solver must be the same on the communicator of the decomposition. Additionally, the user can specify whether the coarse operator is supposed to be symmetric, with the character `'S'`, or not, with the character `'G'`.

```
HPDDM::Schwarz<SUBDOMAIN, COARSEOPERATOR, 'G', double> K;
```

Algorithm 2.14: Instantiation of the preconditioner.

The communication buffers should now be initialized inside the preconditioner, and thereafter, the local partition of unity can be computed.

```
K.Subdomain::initialize(A, o.cbegin(), o.cend(), mapping);
for(std::vector<int>* pt : mapping)
   delete pt;
K.multiplicityScaling(d);
K.initialize(d);
```

Algorithm 2.15: Initialization of the internal data structures.

It is now time to factorize the local matrices $\{A_{ii}\}_{i=1}^{N}$ as well as assemble, redistribute, and factorize the coarse operator.

```
K.callNumfact();
std::vector<unsigned short> parm(5);
parm[HPDDM::P] = 1;
parm[HPDDM::TOPOLOGY] = 0;
parm[HPDDM::DISTRIBUTION] = HPDDM::DMatrix::NON_DISTRIBUTED;
parm[HPDDM::STRATEGY] = 3;
parm[HPDDM::NU] = 1;
K.setVectors(deflation);
K.super::initialize(1);
K.buildTwo(MPI_COMM_WORLD, parm);
```

Algorithm 2.16: Factorization of the local linear systems and of the coarse operator.

Eventually, a preconditioned iterative method may now be called to solve the initial problem.

```
unsigned short it = 100;
unsigned short restart = 30;
HPDDM::IterativeMethod::GMRES(K, sol, f, restart, it, stop, K.getCommunicator(), ↴
    ↳ rankWorld == 0 ? 1 : 0);
```

Algorithm 2.17: Use of the preconditioner in the GMRES method.

Chapter 3

Finite element languages

How to link the framework presented in chapter 2 with existing software for the discretization of partial differential equations will now be explained. In section 3.1, FreeFem++ is briefly introduced and the tools needed for performing domain decomposition preconditioning are presented, as in [Jolivet, Dolean, et al. 2012]. In section 3.2, the focus is on Feel++.

Le lien entre la libraire présentée chapitre 2 et des outils existants de discrétisation d'équations aux dérivées partielles est expliqué dans ce chapitre. Dans la section 3.1, FreeFem++ est brièvement introduit et les outils nécessaires aux méthodes de décomposition de domaine sont détaillés, comme dans [Jolivet, Dolean, et al. 2012]. Dans la section 3.2, on se concentre sur Feel++.

Contents

3.1 FreeFem++ 60

3.1.1 Mesh generation and decomposition 61

3.1.2 Matrix assembly 62

3.1.3 Transfer operators 63

3.1.4 Interface with the framework 65

3.1.5 Generating a global numbering 66

3.1.6 Nonlinear and time-dependent solid mechanics 66

3.2 Feel++ 75

3.2.1 Preprocessing steps 76

3.2.2 Transfer operators 77

3.2.3 Retrieving a local matrix 78

3.2.4 Calling the solver 78

Domain decomposition methods are often used in conjunction with Galerkin methods which are used to convert a continuous problem—such as a system of partial differential equations—into a discrete problem that can be solved numerically. From the point of view of a developer, implementing such methods is quite cumbersome when considering high-order approximations, with discontinuous approximation spaces, and unstructured multi-dimensional meshes. However, with the rise of metaprogramming, domain specific languages can be used to efficaciously circumvent such problems. Leveraging comprehensive interfaces with discretization libraries, it is explained in this chapter how to easily use domain decomposition preconditioners.

3.1 FreeFem++

FreeFem++ [Hecht 2012] is a domain-specific language (DSL) that can be used for performing various essential tasks related to the finite element method. While all the following tasks are sequential, they can be called concurrently on different processes.

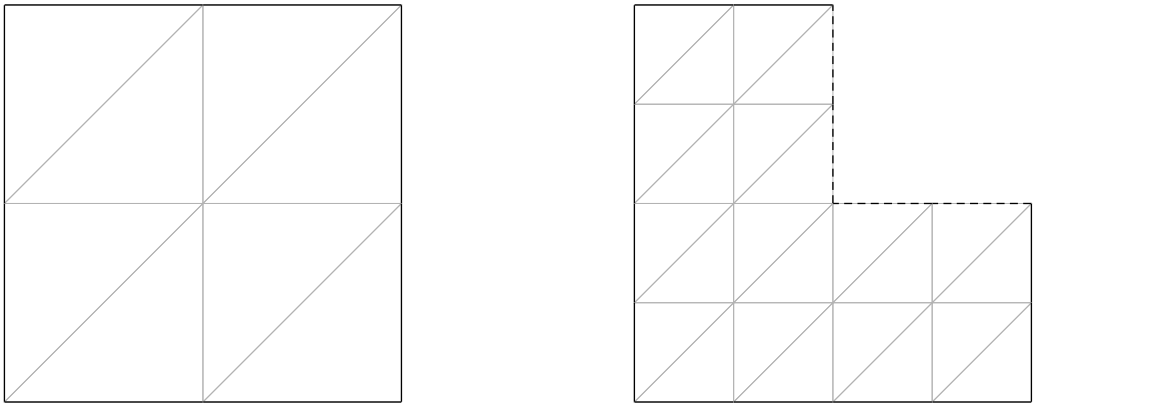
Mesh construction, using user specified geometries, e.g. non-self-intersecting polytopes, plus the capability of "gluing" different meshes using the overloaded operator +.

---

mesh $\mathcal{T}_1(\Omega,$ discretization parameters, label = ...)

It is also be possible to define a mesh by truncating another one, i.e. by removing triangles or tetrahedra, and/or by splitting each triangle or tetrahedron by a given positive integer s, see fig. 3.1.

$\mathcal{T}_2 \leftarrow \mathsf{trunc}(\mathcal{T}_1,$ boolean function to keep or remove elements, split $= s,$
label $= \ldots)$



(a) $\mathcal{T}_1 = \mathsf{square}(2,2)$ **(b)** $\mathcal{T}_2 = \mathsf{trunc}(\mathcal{T}_1, x < 0.5 \,||\, y < 0.5,$ split $= 2,$ label $= \Gamma)$

Fig 3.1: A truncated and refined mesh generated by FreeFem++.
The new boundary edges are labeled Γ (dashed line on the right figure).

Finite element space definition, on an arbitrary mesh $\mathcal{T}$ with various basis functions such as $\mathbb{P}_1$ or $\mathbb{P}_2$ finite elements.

---

fespace $U(\mathcal{T}_1, \mathbb{P}_1)$
fespace $V(\mathcal{T}_1, \mathbb{P}_2)$

Finite element space linear interpolation, from a space U to another space V in a matrix form.

---

matrix $\mathcal{I} \leftarrow \mathsf{interpolate}(U \to V)$

Variational formulation instantiation, with the keyword varf. For eq. (1.5), it would lead to a line of code similar to:

$$\mathsf{varf}\ \mathbb{V}(u,v) \leftarrow \int_{\mathcal{T}} \nabla u \cdot \nabla v - \int_{\mathcal{T}} f\, v + \mathsf{on}(L, u = 0)$$

---

where $u \in U$ is a trial function, $v \in V$ is a test function, and the keyword on imposes penalized boundary conditions on the boundary elements of $\mathcal{T}_1$ labeled L. The

keyword used for two-dimensional integration is int2d and the one used for three-dimensional integration is int3d.

Matrix and vector assembly, when the finite element spaces for the trial functions and for the test functions are defined, the matrix form (resp. right-hand side) of the varf $\mathbb{V}$ may be computed and should be stored in a sparse format (resp. contiguous block of memory).

matrix $A \leftarrow \mathbb{V}(U, V)$
vector $f \leftarrow \mathbb{V}(0, V)$

Using these keywords, it is possible to perform large-scale experiments even if the FreeFem++ finite element kernel is sequential. Indeed, while there exists a MPI interface inside the language, there is no level of abstraction for performing parallel mesh construction, matrix assembly, etc. As such, vectors generated by FreeFem++ are simple containers that encapsulate pointers to scalars (double-precision floating-point reals or complexes), and matrices are represented in CSR format, cf. section 2.1.2. This could be seen as a major setback for parallel computing, but the approach that will be described in this section has proven to be very effective.

3.1.1 Mesh generation and decomposition

The obvious initial step as in all finite element method numerical simulations is to load or discretize a geometry. This can be done using built-in tools as described in the previous paragraph, but also by loading meshes generated offline using a third-party software such as Gmsh [Geuzaine and Remacle 2009] or TetGen [Si 2013]. Thanks to the ability of FreeFem++ to refine any given mesh, cf. fig. 3.1, it is possible to start with a somehow coarse global mesh that will be refined after the partitioning step. From the point of view of the finite element method, a partitioning is a discontinuous piecewise (with respect to $\mathcal{T}$) constant $\mathbb{P}_0$ function p, meaning that each element of $\mathcal{T}$ is given a constant value $i \in [\![1; N]\!]$ that will correspond to the subdomain Ω_i^0 to which it belongs, formally: $\Omega_i^0 = p^{-1}(i)$.

User-defined partitioning If the shape of Ω is regular with some symmetry properties (e.g. rectangles in $\mathbb{R}^2$, cuboids in $\mathbb{R}^3$), an analytical partitioning of Ω can be defined. As an example, the partitionings used in some numerical simulations of chapter 5 are given below.

Algorithm 3.2: Partitioning of the unit square.

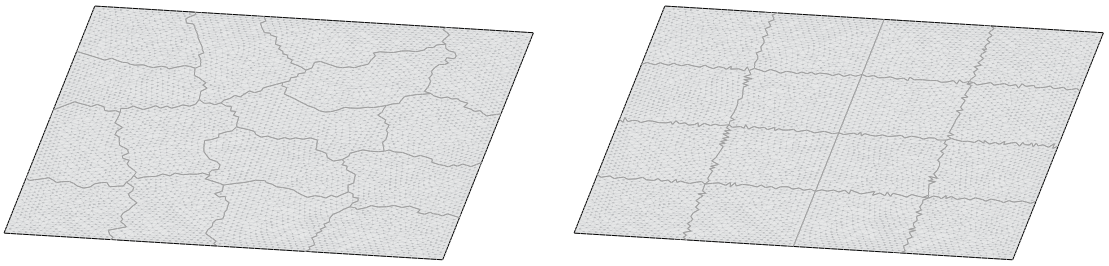
Input: mesh $\mathcal{T}$, number of subdomains N
$i \leftarrow \lfloor \sqrt{N} \rfloor$
while $N \not\equiv 0 \mod i$ **do**
  $i \leftarrow i - 1$
$j \leftarrow \dfrac{N}{i}$
fespace $P(\mathcal{T}, \mathbb{P}_0)$
$Pp \leftarrow \lfloor \mathsf{x} \rfloor j + \lfloor \mathsf{y} \rfloor$

In algorithms 3.2 and 3.3, x, y and z are $\mathbb{P}_0$ functions that return the coordinates of the point at which it is being evaluated.

Algorithm 3.3: Partitioning of the unit cube.

Input: mesh $\mathcal{T}$, number of subdomains N
$i \leftarrow \lfloor \sqrt[3]{N} \rfloor$
while $N \not\equiv 0 \mod i$ **do**
  $i \leftarrow i - 1$
$j \leftarrow \left\lfloor \sqrt{\dfrac{N}{i}} \right\rfloor$
while $\dfrac{N}{i} \not\equiv 0 \mod j$ **do**
  $j \leftarrow j - 1$
$k \leftarrow \left\lfloor \dfrac{N}{ij} \right\rfloor$
fespace $P(\mathcal{T}, \mathbb{P}_0)$
$Pp \leftarrow \lfloor \mathsf{x} \rfloor jk + \lfloor \mathsf{y} \rfloor k + \lfloor \mathsf{z} \rfloor$

Algorithmic graph partitioning When Ω defines a complex geometry, it is best to use graph partitioners as introduced section 2.1.5 that will lead to more balanced decompositions: less communication, similar workload and such. These partitioners can be directly called from within FreeFem++ DSL.



(a) With a graph partitioner **(b)** With the user-defined partitioning algorithm 3.2

Fig 3.4: Examples of sixteen-way partitionings of $\Omega = [0;1]^2$.

User-supplied partitioning Another alternative is to directly load a previously partitioned mesh. For example, using a computer-aided design software, it is possible to mesh complex geometries and split them into multiple parts. This results in a much faster preprocessing as it is now unnecessary to build the global mesh $\mathcal{T}$ and the global $\mathbb{P}_0$ finite element space on a single MPI process. The only two drawbacks are the fact that it is a little more Input/Output intensive and that the list of neighboring subdomains $\{\mathcal{O}_i\}_{i=1}^N$ must also be user-supplied.

$\mathcal{T}_i^0 \leftarrow \mathsf{trunc}(\mathcal{T}, |p-i| < 10^{-6}, \mathrm{label} = 10)$ `// local operations`

Using the nonoverlapping meshes, each subdomain builds an axis-aligned minimum bounding box that encloses $\mathcal{T}_i^0$, for $i \in [\![1;N]\!]$. Each dimension of the boxes is increased by $2 \cdot l \cdot \max_{\tau \in \mathcal{T}_i^0} h_\tau$ (where h_τ is the characteristic size of element $\tau \in \mathcal{T}$) and the global mesh is now replaced by all elements enclosed by the local box.

$\mathcal{T} \leftarrow \mathsf{trunc}(\mathcal{T}, x \text{ and } y \text{ and } z \text{ are coordinates inside the local box})$

This is only done in order to free up memory and speed up preprocessing steps that need information about neighboring subdomains. In particular, for building the overlapping decomposition, each process can now define the $\mathbb{P}_0$ and $\mathbb{P}_1$ finite element spaces on $\mathcal{T}$. This is possible because the global mesh $\mathcal{T}$ has been shrunk locally to fit to each $\left\{\mathcal{T}_i^l\right\}_{i=1}^N$. For $i \in [\![1;N]\!]$, on the one hand, a local function $\mathbb{1}_i^l$ is defined recursively on the discontinuous piecewise constant finite element space as such:

$$\mathbb{1}_i^l = \begin{cases} 1 & \text{on all elements } \tau \in \mathcal{T}_i^l \\ 0 & \text{otherwise} \end{cases}$$

On the other hand, the local function $\widetilde{\chi}_i^l$ introduced eq. (1.6) may also be defined. At this point in the algorithm, the global mesh $\mathcal{T}$ can be completely freed from memory on each process, and the local subdomain can be refined concurrently.

3.1.2 Matrix assembly

When the level of overlap is greater than zero, i.e. $l > 0$, the local matrices $\{A_{ii}\}_{i=1}^N$ defined in eq. (1.11) must be assembled as they are needed for one-level methods. They might seem easy to compute using the variational formulation eq. (1.1) on each subdomain $\{\Omega_i\}_{i=1}^N$ but

it is not that simple since this approach would not take into account the contributions of the integral outside of the subdomains from boundary d.o.f. Instead, that would yield the unassembled operators $\left\{\mathring{A}_i\right\}_{i=1}^N$ given definition 1.7 needed for substructuring methods:

$$\begin{aligned}(A_{ii})_{jk_{1\leqslant j,k\leqslant n_i}} &= \sum_{\tau\in\mathcal{T}} a(\phi_{\mathcal{N}_i(k)|_\tau}, \phi_{\mathcal{N}_i(j)|_\tau}) \\ &= \underbrace{\sum_{\tau\in\mathcal{T}_i} a(\phi_{\mathcal{N}_i(k)|_\tau}, \phi_{\mathcal{N}_i(j)|_\tau})}_{=\left(\mathring{A}_i\right)_{jk}} + \underbrace{\sum_{\tau\in\mathcal{T}\setminus\mathcal{T}_i} a(\phi_{\mathcal{N}_i(k)|_\tau}, \phi_{\mathcal{N}_i(j)|_\tau})}_{\neq 0 \implies (j,k)\in\mathcal{N}_b^{(i)}\times\mathcal{N}_b^{(i)}}.\end{aligned}$$

As described in algorithm 3.5, one approach to build the correct local matrix on each subdomain is the following:

1. instantiate the variational formulation on $\mathcal{T}^{l+1}$ instead of $\mathcal{T}^l$ and assemble the corresponding formulation in a matrix A^+,
2. build the linear interpolator of the finite element space associated with $\mathcal{T}^{l+1}$ to the one associated with $\mathcal{T}^l$ and store the resulting matrix in $\mathcal{I}$,
3. compute $A_{ii} = \mathcal{I}A^+\mathcal{I}^T$, which amounts to deleting in A^+ rows and columns associated with $\Omega^{l+1}\setminus\Omega^l$.

The local right-hand side is obtained using the same technique.

Algorithm 3.5: Construction of the local matrix and local right-hand side.

Input: meshes $\mathcal{T}_i^l$ and $\mathcal{T}_{i+1}^l$, finite element $\mathbb{P}$
1 fespace $V_i(\mathcal{T}_i^l, \mathbb{P})$
2 fespace $W_i(\mathcal{T}_i^{l+1}, \mathbb{P})$
3 matrix $\mathcal{I} \leftarrow$ interpolate$(W_i \to V_i)$
4 matrix $A^+ \leftarrow \mathbb{V}(W_i, W_i)$
5 matrix $A_{ii} \leftarrow \mathcal{I}A^+\mathcal{I}^T$
6 vector $g \leftarrow \mathbb{V}(0, W_i)$
7 vector $f \leftarrow \mathcal{I}g$

In practice, line 5 of the previous algorithm is computed using a much simpler and more efficient way than actually performing a double sparse matrix-sparse matrix product. If the number of d.o.f. of each $\{W_i\}_{i=1}^N$ is $\left\{n_i^+\right\}_{i=1}^N$, then this is achieved by copying into A_{ii} only nonzero entries of A^+ whose row and column indices $(j,k) \in [\![1; n_i^+]\!]^2$ are associated with at least one nonzero entry of $\mathcal{I}$, i.e. if there exists $m \in [\![1; n_i]\!]$ such that $(\mathcal{I})_{mj} \neq 0$ or $(\mathcal{I})_{mk} \neq 0$.

When the level of overlap is zero and substructuring methods are being used, the assembly of the local unassembled matrices are trivial using a variational formulation inside each subdomain $\{\Omega_i\}_{i=1}^N$.

3.1.3 Transfer operators

For overlapping methods, an automatic and concurrent way of computing the transfer operators on each subdomain $i \in [\![1; N]\!]$, $\left\{R_i R_j^T\right\}_{j\in\mathcal{O}_i}$ is now explained. What is needed is the structure `mapping` used in algorithm 2.2. Unlike in algorithm 2.11, there is no way to

provide an algebraic construction since this time the decomposition is not known a priori. Instead, the construction relies on the finite element interpolator.

Algorithm 3.6: Construction of the overlapping transfer operators.

Input: mesh $\mathcal{T}_i^l$, set $\mathcal{O}_i$, local partition of unity χ_i
foreach $k \in \mathcal{O}_i$ **do**
  mesh $\mathcal{T}_{\text{intersection}} \leftarrow \mathsf{trunc}(\mathcal{T}_i^l, \chi_{i|\Omega_i^l \cap \Omega_k^l} > 0)$
  fespace $V_{\text{intersection}}(\mathcal{T}_{\text{intersection}}, \mathbb{P})$
  matrix $R_k \leftarrow \mathsf{interpolate}(V_{\text{intersection}} \rightarrow V_i)$
  `mapping`$[\mathcal{O}_i^{-1}(k)].second =$ column indices of R_k

Two major drawbacks of FreeFem++ are that:

1. one-dimensional meshes are not supported. When dealing with substructuring methods in two dimensions, it is however of paramount importance since the dimension of the interfaces between subdomains is one, and,

2. assembling interpolation matrices between finite element spaces of different geometric dimensions is not possible, but it is once again necessary for substructuring methods when defining interfaces of subdomains.

Keeping these limitations in mind, the approach described next is useful for computing the transfer operators on each subdomain $i \in [\![1; N]\!]$: $\left\{ B^{(i)^T} B^{(j)} \right\}_{\substack{1 \leqslant i \leqslant N \\ j \in \overline{\mathcal{O}}_i}}$ for the BDD method, or $\left\{ \underline{B}^{(i)^T} \underline{B}^{(j)} \right\}_{\substack{1 \leqslant i \leqslant N \\ j \in \overline{\mathcal{O}}_i}}$ for the FETI method. Because the geometric dimension has to stay the same, the first step is to build a decomposition with a minimum level of overlap equal to one. It will be used to determine the true interfaces between subdomains which are of lower geometric dimensions. For that matter, let $\{\mathcal{T}_i^1\}_{i=1}^N$ be the overlapping decomposition and $\{\chi_i\}_{i=1}^N$ be the partition of unity. It is assumed next that for each nonoverlapping mesh $\mathcal{T}_i$, $i \in [\![1; N]\!]$, the artificial boundary faces (i.e. those that do not have any prescribed boundary conditions and that are the result of the subdomains tearing) are labeled by a fixed integer, e.g. 10.

Algorithm 3.7: Construction of the nonoverlapping transfer operators.

Input: meshes $\mathcal{T}_i$ and $\mathcal{T}_i^1$, set $\mathcal{O}_i$, local partition of unity χ_i
fespace $U_i(\mathcal{T}_i, \mathbb{P})$
varf $\mathbb{V}(u, v) \leftarrow \mathsf{on}(10, u = 1)$
vector $f \leftarrow \mathbb{V}(0, U)$
foreach $k \in \mathcal{O}_i$ **do**
  mesh $\mathcal{T}_{\text{intersection}} \leftarrow \mathsf{trunc}(\mathcal{T}_i^1, \chi_{i|\Omega_i^1 \cap \Omega_k^1} > 0)$
  fespace $V_{\text{intersection}}(\mathcal{T}_{\text{intersection}}, \mathbb{P})$
  matrix $R_k \leftarrow \mathsf{interpolate}(V_{\text{intersection}} \rightarrow U_i)$
  int $ndof_k \leftarrow$ number of d.o.f. in $V_{\text{intersection}}$
  vector $g_k \leftarrow R_k^T f$
  `MPI_Isend`$(g_k, ndof_k, \texttt{MPI\_DOUBLE}, k, rq_1[k])$
  `MPI_Irecv`$(h_k, ndof_k, \texttt{MPI\_DOUBLE}, k, rq_2[k])$
foreach $k \in \mathcal{O}_i$ **do**
  `MPI_Waitany`$(\#\mathcal{O}_i, rq_1, \&\text{index})$
  int $m = \mathcal{O}_i[\text{index}]$
  for $l \leftarrow 1$ **to** $ndof_m$ **do**
    if $g_m[l] = 1$ **and** $h_m[l] = 1$ **then**
      `mapping`$[\mathcal{O}_i^{-1}(m)].second$.emplace_back$(l)$
`MPI_Waitall`$(\#\mathcal{O}_i, rq_2)$

3.1.4 Interface with the framework

Each and every object of the framework described in section 2.2.1 can now be initialized with the structures computed with FreeFem++. For better readability, three objects have been added to the DSL: schwarz, bdd, and feti and they can be initialized inside a FreeFem++ script using the syntax given below.

$$\mathsf{schwarz}\ A\left(A_{ii}, \mathcal{O}_i, \left\{R_i R_j^T\right\}_{j \in \mathcal{O}_i}, \mathsf{scaling} = D, \mathsf{communicator} = comm\right)$$

This must be a collective call on the MPI communicator $comm$ which is an optional argument that defaults to `MPI_COMM_WORLD`. In FreeFem++, the set of transfer operators is stored as an array of arrays of integer indices, but it is preprocessed by the framework, alongside $\mathcal{O}_i$, so that internally the structure `mapping` defined eq. (2.3) is filled correctly. The call for initializing an object of type bdd or feti is the same, except that the user should have computed the local unassembled operators.

Attaching a coarse operator

When using either overlapping or nonoverlapping methods, it is possible to attach an abstract coarse operator using user-supplied local deflation vectors. With nonoverlapping methods, it is also possible to let the framework build a coarse operator using the appropriate eigenvectors of eq. (1.43). Since the generalized eigenvalue problem only relies on operators already partially needed by the BDD method or the FETI method, there is no additional computation needed by FreeFem++. With overlapping methods, the appropriate eigenvectors of eq. (1.40) may be used to build a coarse operator. In that case, the user has to provide the unassembled operators $\left\{\mathring{A}_i\right\}_{i=1}^N$ which are a priori not needed by such methods, but which can be computed easily using FreeFem++, cf. section 3.1.2.

It is then possible to solve the linear system using the appropriate keyword added to FreeFem++ DSL.

3.1.5 Generating a global numbering

While all previous distributed algorithms do not require a global numbering of the finite element space or a sophisticated data structure for storing distributed vectors and matrices, it might still be useful to compute such numbering. In particular, if one were to use for example PETSc [Balay, Gropp, et al. 1997] which uses a row-wise distribution, this would be mandatory. Hence, an algorithm for generating a global numbering will now be presented, and it will be extensively used in section 5.2 when comparing the framework against other state of the art iterative solvers interfaced with PETSc. It is based on a single sweep method with ascending order of process ranks. For any given process $i \in [\![1; N]\!]$:

1. if $i = 0$, then set $start$ to 0, otherwise, receive $start$ from process $i - 1$,
2. $\forall j \in \mathcal{O}_i \ : \ j < i$, receive the global numbering of each neighbor with lower ranks computed on the overlaps $\{R_i R_j^T\}$,
3. number all local d.o.f., using a $start$-based numbering, that are inside the subdomain and that are not duplicated on processes with lower ranks, $R_i \sum_{j \in \overline{\mathcal{O}}_i \, : \, j \geqslant i} R_j^T$, and set the last index to end,
4. $\forall j \in \mathcal{O}_i \ : \ j > i$, send the global numbering of each neighbor with higher ranks computed on the overlap $\{R_i R_j^T\}$,
5. if $i \neq N - 1$, then send end to process $i + 1$,
6. when the global number of d.o.f n is needed by all processes, then, on process $N - 1$, set n to end and initiate the following collective communication:

$$\texttt{MPI\_Bcast(\&}n\texttt{, 1, MPI\_UNSIGNED, }N\texttt{ - 1, comm).}$$

3.1.6 Nonlinear and time-dependent solid mechanics

In section 5.1, the system of linear elasticity will be introduced eq. (5.1). Linear elasticity is a simplification of the more general nonlinear theory of elasticity where strains are infinitesimal and there exists linear relationships between the components of stress and strain. In this paragraph, hyperelasticity will be studied. FreeFem++ can be used to easily manipulate nonlinear operators by using macros that will mimic automatic differentiation. Let Ω be a tridimensional reference configuration. The equation of static equilibrium is then in the Lagrangian formalism, for $u \in [H_0^1(\Omega)]^3$:

$$-\nabla \cdot ((I + \nabla u)\, \Sigma) = f, \tag{3.1}$$

with suitable boundary conditions, where u is the displacement vector, Σ is the second Piola-Kirchhoff stress tensor, and f are body forces. The weak formulation of eq. (3.1) is,

for all test functions $v \in \left[H_0^1(\Omega)\right]^3$:

$$\begin{aligned} -\int_\Omega (I + \nabla u)\, \Sigma : \nabla v &= \int_\Omega l(v) \\ -\int_\Omega \Sigma : \left(I + \nabla u^T\right) \nabla v &= \int_\Omega l(v) \\ -\int_\Omega \Sigma : D\varepsilon(u)[v] &= \int_\Omega l(v), \end{aligned}$$

where ε is the Green–Lagrange strain tensor defined as:

$$\varepsilon(u) = \frac{1}{2}\left(\underbrace{\nabla u + \nabla u^T}_{=2\varepsilon_{\mathrm{L}}(u)} + \underbrace{\nabla u^T \nabla u}_{=2\varepsilon_{\mathrm{NL}}(u)}\right). \tag{3.2}$$

ε_{L} is the linearized strain tensor:

$$\varepsilon_{\mathrm{L}}(u) = \frac{1}{2}\left(\nabla u + \nabla u^T\right).$$

Indeed, since Σ is symmetric,

$$\begin{aligned} \Sigma : \left(I + \nabla u^T\right) \nabla v &= \frac{1}{2}\Sigma : \left(\left(I + \nabla u^T\right) \nabla v + \nabla v^T \left(I + \nabla u\right)\right) \\ &= \frac{1}{2}\Sigma : \left(\nabla v + \nabla v^T + \nabla u^T \nabla v + \nabla v^T \nabla u\right) \\ &= \Sigma : D\varepsilon(u)[v]\,. \end{aligned} \tag{3.3}$$

Note that the Gâteaux derivative of ε at $u \in \Omega$ in the direction $v \in \Omega$ is:

$$D\varepsilon(u)[v] = \varepsilon_{\mathrm{L}}(v) + D\varepsilon_{\mathrm{NL}}(u)[v]\,.$$

Within the context of hyperelasticity, it is assumed that the stress-strain relationship derives from a stored energy density function $\mathcal{W}$ such that:

$$\Sigma = \frac{\partial \mathcal{W}}{\partial \varepsilon}.$$

The total strain energy is then given by:

$$\Pi(u) = \int_\Omega \mathcal{W}(\varepsilon(u))\,,$$

and one has:

$$D\Pi(u)[v] = \int_\Omega \frac{\partial \mathcal{W}}{\partial \varepsilon} : D\varepsilon(u)[v]\,,$$

and,

$$D^2\Pi(u)[v,w] = \int_\Omega \frac{\partial^2 \mathcal{W}}{\partial \varepsilon^2} : D\varepsilon(u)[w] : D\varepsilon(u)[v] + \int_\Omega \frac{\partial \mathcal{W}}{\partial \varepsilon} : D^2\varepsilon(u)[v,w]\,.$$

In its most general form, the relationship between Σ and ε can be nonlinear. In order to apply a Newton–Rhapson solution process, this relationship needs to be linearized with respect to u in the direction w. Using the chain rule, one has:

$$D\Sigma(\varepsilon(u))[w] = \frac{\partial \Sigma}{\partial \varepsilon}\left(D\varepsilon(u)[w]\right) = \frac{\partial^2 \mathcal{W}}{\partial \varepsilon^2}\left(D\varepsilon(u)[w]\right).$$

Because:

$$\int_\Omega (I+\nabla u+\nabla w)\,\Sigma : \nabla v = \int_\Omega (I+\nabla u+\nabla w) \times \Sigma(\varepsilon(u)+\varepsilon(w)\frac{1}{2}(\nabla w^T\nabla u+\nabla u^T\nabla w))) : \nabla v\,,$$

the linearization with respect to w yields

$$\begin{aligned}\int_\Omega \big((I+\nabla u+\nabla w)\Sigma-(I+\nabla u)\Sigma\big) : \nabla v &= \int_\Omega (I+\nabla u)\,D\Sigma(\varepsilon(u))[w] : \nabla v\\ &+ \int_\Omega \nabla w\;\Sigma(\varepsilon(u)) : \nabla v\\ &= \int_\Omega \frac{\partial^2\mathcal{W}}{\partial\varepsilon^2}(D\varepsilon(u)[w]) : \left(I+\nabla u^T\right)\nabla v\\ &+ \int_\Omega \Sigma(\varepsilon(u)) : D^2\varepsilon(u)[v,w]\\ &= \int_\Omega \mathcal{C} : D\varepsilon(u)[w] : D\varepsilon(u)[v]\\ &+ \int_\Omega \Sigma(\varepsilon(u)) : D^2\varepsilon(u)[v,w]\,,\end{aligned}$$

where:

$$\begin{aligned}D^2\varepsilon(u)[v,w] &= \frac{1}{2}\left(\nabla w^T\nabla v+\nabla v^T\nabla w\right)\\ &= D\varepsilon_{\text{NL}}(v)[w] = D\varepsilon_{\text{NL}}(w)[v]\,,\end{aligned}$$

and $\mathcal{C}$ is the four order symmetric tensor defined as:

$$\mathcal{C}_{ijkl} = \frac{1}{2}\left(\left(\frac{\partial^2\mathcal{W}}{\partial\varepsilon^2}\right)_{ijkl}+\left(\frac{\partial^2\mathcal{W}}{\partial\varepsilon^2}\right)_{jikl}\right). \tag{3.4}$$

In its most general form, $\mathcal{C}$ has $3\times3\times3\times3=81$ coefficients in three dimensions. However, thanks to eq. (3.4) and the symmetry of the strain tensor ε,

$$\mathcal{C}_{ijkl} = \mathcal{C}_{jikl} \tag{3.5a}$$
$$\mathcal{C}_{ijkl} = \mathcal{C}_{ijlk} \tag{3.5b}$$
$$\mathcal{C}_{ijkl} = \mathcal{C}_{klij}\,, \tag{3.5c}$$

so that the number of coefficients can be reduced to $6\times3\times3$, then 6×6, and eventually, 21, cf. eq. (3.10).

To sum up this section, when given a specific stored energy function $\mathcal{W}$, one can compute Σ and $\mathcal{C}$ which yield the stiffness matrix:

$$\int_\Omega \mathbb{K}(u)[v] = -\int_\Omega \Sigma(\varepsilon(u)) : D\varepsilon(u)[v]\;\mathrm{d}v - \int_\Omega l(v)\;\mathrm{d}v,$$

and the tangent stiffness matrix relative to a direction w:

$$\int_\Omega \mathbb{K}'(u)[v,w] := \int_\Omega \mathcal{C} : D\varepsilon(u)[w] : D\varepsilon(u)[v] + \Sigma(\varepsilon(u)) : D^2\varepsilon(u)[v,w]\;\mathrm{d}v.$$

A simple Newton–Rhapson algorithm reads:

1. initialize u_0,
2. loop on n,
   (a) find the solution d_n of:
   $$\int_\Omega \mathbb{K}'(u_n)[v, d_n] = \int_\Omega \mathbb{K}(u_n)[v]\,, \tag{3.6}$$
   (b) find an appropriate length step α_n using for example the backtracking line search:
      i. set $j = 0$, $\alpha_{n_0} = 1$, $\tau = 0.5$,
      ii. while the Armijo–Goldstein condition,
      $$\Pi(u_n + \alpha_{n_j} d_n) \leqslant \Pi(u_n) + c_1 \alpha_{n_j} d_n^T \nabla\Pi(u_n)\,,$$
      is not satisfied, set $\alpha_{n_{j+1}} = \tau\alpha_{n_j}$ and increment j by 1.
   (c) update the displacement $u_{n+1} = u_n - \alpha_n d_n$,
   (d) if $||d_n||$ or $\alpha_n < \varepsilon$, break the loop.

Some hyperelastic material models

The simplest hyperelastic material model is the Saint Venant–Kirchhoff model for which the stored energy function is:

$$\begin{aligned}\mathcal{W} &= \frac{\lambda}{2}(\text{tr}\varepsilon)^2 + \mu\text{tr}(\varepsilon^2)\\ &= \frac{\lambda}{2}\left(\varepsilon_{11} + \varepsilon_{22} + \varepsilon_{33}\right)^2\\ &+ \mu\left(\varepsilon_{11}^2 + \varepsilon_{22}^2 + \varepsilon_{33}^2 + 2\varepsilon_{23}^2 + 2\varepsilon_{31}^2 + 2\varepsilon_{12}^2\right).\end{aligned}$$

This yields:

$$\begin{aligned}\Sigma &= \lambda\text{tr}(\varepsilon)I + 2\mu\varepsilon\\ \mathcal{C}_{ijkl} &= \lambda\delta_{ij}\delta_{kl} + \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}).\end{aligned} \tag{3.7}$$

In order to present the next material model, another important tensor has to be introduced. The right Cauchy–Green deformation tensor is defined as:

$$C(u) := (I + \nabla u)^T (I + \nabla u)\,,$$

so that:

$$C = 2\varepsilon + I \quad \text{and} \quad \frac{\partial\mathcal{W}}{\partial\varepsilon} = 2\frac{\partial\mathcal{W}}{\partial C}\,. \tag{3.8}$$

The invariants of this tensor are:

$$\begin{aligned}I_C &= \text{tr}C\\ II_C &= \frac{1}{2}\left((\text{tr}C)^2 - \text{tr}(C^2)\right)\\ III_C &= \det C = J^2\,.\end{aligned}$$

The Mooney–Rivlin [Mooney 1940; Rivlin 1948] model has a stored energy function of the form:

$$\mathcal{W} = c_{10}(I_C^\star - 3) + c_{01}(II_C^\star - 3) + d_1(J-1)^2 + d_2 \ln J,$$

where:

$$\begin{aligned} I_C^\star &= J^{-2/3} I_C \\ II_C^\star &= J^{-4/3} II_C \,. \end{aligned}$$

The following partial derivatives are needed to compute Σ:

$$\begin{aligned} \frac{\partial I_C}{\partial C} &= I \\ \frac{\partial II_C}{\partial C} &= I_C I - C^T \\ \frac{\partial III_C}{\partial C} &= III_C C^{-T}. \end{aligned}$$

Then,

$$\begin{aligned} \frac{\partial I_C^\star}{\partial C} &= III_C^{-1/3}\left(I - \frac{1}{3} I_C C^{-T}\right) \\ &= III_C^{-1/3}\left(\frac{\partial I_C}{\partial C} - \frac{I_C}{3III_C}\frac{\partial III_C}{\partial C}\right) \\ \frac{\partial II_C^\star}{\partial C} &= III_C^{-2/3}\left(I_C I - C^T - \frac{2}{3} II_C C^{-T}\right) \\ &= III_C^{-2/3}\left(\frac{\partial II_C}{\partial C} - \frac{2II_C}{3III_C}\frac{\partial III_C}{\partial C}\right) \\ \frac{\partial J}{\partial C} &= \frac{1}{2} III_C^{1/2} C^{-T} \\ &= \frac{1}{2} III_C^{-1/2}\frac{\partial III_C}{\partial C}\,, \end{aligned}$$

and,

$$\frac{\partial \mathcal{W}}{\partial I_C^\star} = c_{10} \qquad \frac{\partial \mathcal{W}}{\partial II_C^\star} = c_{01} \qquad \frac{\partial \mathcal{W}}{\partial J} = 2d_1(J-1) + d_2 J^{-1}\,,$$

so that:

$$\begin{aligned} \Sigma &= 2\frac{\partial \mathcal{W}}{\partial I_C^\star}\frac{\partial I_C^\star}{\partial C} + 2\frac{\partial \mathcal{W}}{\partial II_C^\star}\frac{\partial II_C^\star}{\partial C} + 2\frac{\partial \mathcal{W}}{\partial J}\frac{\partial J}{\partial C} \\ &= 2c_{10}\frac{\partial I_C^\star}{\partial C} + 2c_{01}\frac{\partial II_C^\star}{\partial C} + \left(2d_1(J^2 - J) + d_2\right) C^{-T}. \end{aligned}$$

The following fourth order tensor are then needed to compute $\mathcal{C}$:

$$\frac{\partial^2 I_C}{\partial C_{ij}\partial C_{kl}} = 0 \tag{3.9a}$$

$$\frac{\partial^2 II_C}{\partial C_{ij}\partial C_{kl}} = \delta_{ij}\delta_{kl} - \delta_{jk}\delta_{il} \tag{3.9b}$$

$$\frac{\partial^2 III_C}{\partial C_{ij}\partial C_{kl}} = III_C\left(C^{-1}_{ji}C^{-1}_{lk} - C^{-1}_{li}C^{-1}_{jk}\right). \tag{3.9c}$$

Equation (3.9c) comes from the equality:

$$\frac{\partial A^{-1}_{ij}}{\partial A_{kl}} = -A^{-1}_{ik}A^{-1}_{lj} \implies \frac{\partial A^{-1}_{ji}}{\partial A_{kl}} = -A^{-1}_{li}A^{-1}_{jk}\,.$$

Hence,

$$\begin{aligned}
\frac{\partial^2 I^\star_C}{\partial C_{ij}\partial C_{kl}} &= III_C^{-1/3}\left(\frac{I_C}{9}C^{-1}_{ji}C^{-1}_{lk} - \frac{1}{3}\left(C^{-1}_{ji}\delta_{kl} + \delta_{ij}C^{-1}_{lk}\right) + \frac{I_C}{3}C^{-1}_{li}C^{-1}_{jk}\right)\\
&= III_C^{-1/3}\left(\frac{4I_C}{9III_C^2}\frac{\partial III_C}{\partial C}\otimes\frac{\partial III_C}{\partial C} - \frac{1}{3III_C}\left(\frac{\partial III_C}{\partial C}\otimes\frac{\partial I_C}{\partial C} + \frac{\partial I_C}{\partial C}\otimes\frac{\partial III_C}{\partial C}\right)\right.\\
&\quad\left. - \frac{I_C}{3III_C}\frac{\partial^2 III_C}{\partial C^2}\right)\\
\frac{\partial^2 II^\star_C}{\partial C_{ij}\partial C_{kl}} &= III_C^{-2/3}\left(\delta_{ij}\delta_{kl} - \delta_{jk}\delta_{il} + \frac{10II_C}{9}C^{-1}_{ji}C^{-1}_{lk}\right.\\
&\quad\left. + \frac{2}{3}\left(C^{-1}_{ji}(I_C\delta_{jk} - C_{kj}) + (I_C\delta_{ij} - C_{ji})C^{-1}_{lk}\right) - \frac{2II_C}{3}\left(C^{-1}_{ji}C^{-1}_{lk} - C^{-1}_{li}C^{-1}_{jk}\right)\right)\\
&= III_C^{-2/3}\left(\frac{\partial^2 II_C}{\partial C^2} + \frac{10II_C}{9III_C^2}\frac{\partial III_C}{\partial C}\otimes\frac{\partial III_C}{\partial C}\right.\\
&\quad\left. + \frac{2}{3III_C}\left(\frac{\partial III_C}{\partial C}\otimes\frac{\partial II_C}{\partial C} + \frac{\partial II_C}{\partial C}\otimes\frac{\partial III_C}{\partial C}\right) - \frac{2II_C}{3III_C}\frac{\partial III_C^2}{\partial C^2}\right)\\
\frac{\partial^2 J}{\partial C_{ij}\partial C_{kl}} &= \frac{1}{4}III_C^{1/2}\left(C^{-1}_{ji}C^{-1}_{lk} - 2C^{-1}_{li}C^{-1}_{jk}\right)\\
&= \frac{1}{2}III_C^{-1/2}\left(\frac{\partial^2 III_C}{\partial C^2} - \frac{1}{2III_C}\frac{\partial III_C}{\partial C}\otimes\frac{\partial III_C}{\partial C}\right),
\end{aligned}$$

and,

$$\frac{\partial^2 \mathcal{W}}{\partial J^2} = 2d_1 - d_2 J^{-2}\,.$$

Eventually:

$$\begin{aligned}
\left(\frac{\partial^2 \mathcal{W}}{\partial \varepsilon^2}\right)_{ijkl} &= 4\left(\frac{\partial^2 \mathcal{W}}{\partial C^2}\right)_{ijkl}\\
&= 4\frac{\partial \mathcal{W}}{\partial I^\star_C}\frac{\partial^2 I^\star_C}{\partial C^2} + 4\frac{\partial \mathcal{W}}{\partial II^\star_C}\frac{\partial^2 II^\star_C}{\partial C^2} + 4\frac{\partial \mathcal{W}}{\partial J}\frac{\partial^2 J}{\partial C^2} + 4\frac{\partial^2 \mathcal{W}}{\partial J^2}\frac{\partial J}{\partial C}\otimes\frac{\partial J}{\partial C}\\
&= 4c_{10}\frac{\partial^2 I^\star_C}{\partial C_{ij}\partial C_{kl}} + 4c_{01}\frac{\partial^2 II^\star_C}{\partial C_{ij}\partial C_{kl}} + \left(2d_1\left(J^2 - J\right) + 2d_1J^2\right)C^{-1}_{ji}C^{-1}_{lk}\\
&\quad - 4d_1\left(J^2 - J\right)C^{-1}_{li}C^{-1}_{jk} + d_2\left(\left(C^{-1}_{ji}C^{-1}_{lk} - 2C^{-1}_{li}C^{-1}_{jk}\right) - C^{-1}_{ji}C^{-1}_{lk}\right)\\
&= 4c_{10}\frac{\partial^2 I^\star_C}{\partial C_{ij}\partial C_{kl}} + 4c_{01}\frac{\partial^2 II^\star_C}{\partial C_{ij}\partial C_{kl}}\\
&\quad + 2\left(d_1\left(2J^2 - J\right)C^{-1}_{ji}C^{-1}_{lk} - \left(d_2 + 2d_1\left(J^2 - J\right)\right)C^{-1}_{li}C^{-1}_{jk}\right).
\end{aligned}$$

Implementation details

Using the considerations of eq. (3.5), it is only necessary to consider the following terms of the strain tensor: $\varepsilon_{11}, \varepsilon_{22}, \varepsilon_{33}, \varepsilon_{23}, \varepsilon_{31}, \varepsilon_{12}$. Using Voigt notation, $\mathcal{C}$ can be expressed as:

$$\mathcal{C} = \begin{pmatrix} \mathcal{C}_{1111} & \mathcal{C}_{1122} & \mathcal{C}_{1133} & \mathcal{C}_{1123} & \mathcal{C}_{1131} & \mathcal{C}_{1112} \\ \mathcal{C}_{1122} & \mathcal{C}_{2222} & \mathcal{C}_{2233} & \mathcal{C}_{2223} & \mathcal{C}_{2231} & \mathcal{C}_{2212} \\ \mathcal{C}_{1133} & \mathcal{C}_{2233} & \mathcal{C}_{3333} & \mathcal{C}_{3323} & \mathcal{C}_{3331} & \mathcal{C}_{3312} \\ \mathcal{C}_{1123} & \mathcal{C}_{2223} & \mathcal{C}_{3323} & \mathcal{C}_{2323} & \mathcal{C}_{2331} & \mathcal{C}_{2312} \\ \mathcal{C}_{1131} & \mathcal{C}_{2231} & \mathcal{C}_{3331} & \mathcal{C}_{2331} & \mathcal{C}_{3131} & \mathcal{C}_{3112} \\ \mathcal{C}_{1112} & \mathcal{C}_{2212} & \mathcal{C}_{3312} & \mathcal{C}_{2312} & \mathcal{C}_{3112} & \mathcal{C}_{1212} \end{pmatrix}. \tag{3.10}$$

In three dimensions, the displacement vector $u \in [H_0^1(\Omega)]^3$ can be written as the tensor product of three finite element functions, each approximating functions of $H_0^1(\Omega)$, i.e. $u = (u_1, u_2, u_3)$ where $\{u_i = g_i(x_1, x_2, x_3)\}_{i=1}^3$, so that,

$$\nabla u = \begin{pmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} & \frac{\partial u_1}{\partial x_3} \\ \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} \\ \frac{\partial u_3}{\partial x_1} & \frac{\partial u_3}{\partial x_2} & \frac{\partial u_3}{\partial x_3} \end{pmatrix},$$

and,

$$\begin{aligned}
2\varepsilon(u)_{11} &= 2\frac{\partial u_1}{\partial x_1} &&+ \left(\frac{\partial u_1}{\partial x_1}\right)^2 + \left(\frac{\partial u_2}{\partial x_1}\right)^2 + \left(\frac{\partial u_3}{\partial x_1}\right)^2 \\
2\varepsilon(u)_{22} &= 2\frac{\partial u_2}{\partial x_2} &&+ \left(\frac{\partial u_1}{\partial x_2}\right)^2 + \left(\frac{\partial u_2}{\partial x_2}\right)^2 + \left(\frac{\partial u_3}{\partial x_2}\right)^2 \\
2\varepsilon(u)_{33} &= 2\frac{\partial u_3}{\partial x_3} &&+ \left(\frac{\partial u_1}{\partial x_3}\right)^2 + \left(\frac{\partial u_2}{\partial x_3}\right)^2 + \left(\frac{\partial u_3}{\partial x_3}\right)^2 \\
2\varepsilon(u)_{23} &= \frac{\partial u_2}{\partial x_3} + \frac{\partial u_3}{\partial x_2} &&+ \frac{\partial u_1}{\partial x_2}\frac{\partial u_1}{\partial x_3} + \frac{\partial u_2}{\partial x_2}\frac{\partial u_2}{\partial x_3} + \frac{\partial u_3}{\partial x_2}\frac{\partial u_3}{\partial x_3} \\
2\varepsilon(u)_{31} &= \frac{\partial u_3}{\partial x_1} + \frac{\partial u_1}{\partial x_3} &&+ \frac{\partial u_1}{\partial x_1}\frac{\partial u_1}{\partial x_3} + \frac{\partial u_2}{\partial x_1}\frac{\partial u_2}{\partial x_3} + \frac{\partial u_3}{\partial x_1}\frac{\partial u_3}{\partial x_3} \\
2\varepsilon(u)_{12} &= \frac{\partial u_1}{\partial x_2} + \frac{\partial u_2}{\partial x_1} &&+ \frac{\partial u_1}{\partial x_1}\frac{\partial u_1}{\partial x_2} + \frac{\partial u_2}{\partial x_1}\frac{\partial u_2}{\partial x_2} + \frac{\partial u_3}{\partial x_1}\frac{\partial u_3}{\partial x_2}.
\end{aligned}$$

Linear part $\varepsilon_{\text{L}}(u)$ Nonlinear part $\varepsilon_{\text{NL}}(u)$

$$
\begin{aligned}
2D\varepsilon_{\mathrm{NL}}(u)[v]_{11} &= 2\left(\frac{\partial u_1}{\partial x_1}\frac{\partial v_1}{\partial x_1} + \frac{\partial u_2}{\partial x_1}\frac{\partial v_2}{\partial x_1} + \frac{\partial u_3}{\partial x_1}\frac{\partial v_3}{\partial x_1}\right)\\
2D\varepsilon_{\mathrm{NL}}(u)[v]_{22} &= 2\left(\frac{\partial u_1}{\partial x_2}\frac{\partial v_1}{\partial x_2} + \frac{\partial u_2}{\partial x_2}\frac{\partial v_2}{\partial x_2} + \frac{\partial u_3}{\partial x_2}\frac{\partial v_3}{\partial x_2}\right)\\
2D\varepsilon_{\mathrm{NL}}(u)[v]_{33} &= 2\left(\frac{\partial u_1}{\partial x_3}\frac{\partial v_1}{\partial x_3} + \frac{\partial u_2}{\partial x_3}\frac{\partial v_2}{\partial x_3} + \frac{\partial u_3}{\partial x_3}\frac{\partial v_3}{\partial x_3}\right)\\
2D\varepsilon_{\mathrm{NL}}(u)[v]_{23} &= \frac{\partial u_1}{\partial x_2}\frac{\partial v_1}{\partial x_3} + \frac{\partial u_2}{\partial x_2}\frac{\partial v_2}{\partial x_3} + \frac{\partial u_3}{\partial x_2}\frac{\partial v_3}{\partial x_3}\\
&\quad + \frac{\partial u_1}{\partial x_3}\frac{\partial v_1}{\partial x_2} + \frac{\partial u_2}{\partial x_3}\frac{\partial v_2}{\partial x_2} + \frac{\partial u_3}{\partial x_3}\frac{\partial v_3}{\partial x_2}\\
2D\varepsilon_{\mathrm{NL}}(u)[v]_{31} &= \frac{\partial u_1}{\partial x_3}\frac{\partial v_1}{\partial x_1} + \frac{\partial u_2}{\partial x_3}\frac{\partial v_2}{\partial x_1} + \frac{\partial u_3}{\partial x_3}\frac{\partial v_3}{\partial x_1}\\
&\quad + \frac{\partial u_1}{\partial x_1}\frac{\partial v_1}{\partial x_3} + \frac{\partial u_2}{\partial x_1}\frac{\partial v_2}{\partial x_3} + \frac{\partial u_3}{\partial x_1}\frac{\partial v_3}{\partial x_3}\\
2D\varepsilon_{\mathrm{NL}}(u)[v]_{12} &= \frac{\partial u_1}{\partial x_1}\frac{\partial v_1}{\partial x_2} + \frac{\partial u_2}{\partial x_1}\frac{\partial v_2}{\partial x_2} + \frac{\partial u_3}{\partial x_1}\frac{\partial v_3}{\partial x_2}\\
&\quad + \frac{\partial u_1}{\partial x_2}\frac{\partial v_1}{\partial x_1} + \frac{\partial u_2}{\partial x_2}\frac{\partial v_2}{\partial x_1} + \frac{\partial u_3}{\partial x_2}\frac{\partial v_3}{\partial x_1}.
\end{aligned}
$$

These functions may be computed using FreeFem++ macros, that are similar to macros interpreted by a compiler preprocessor.

```
macro EL(u)[dx(u#1),
            dy(u#2),
            dz(u#3),
            dz(u#2) + dy(u#3),
            dx(u#3) + dz(u#1) ,
            dy(u#1) + dx(u#2)]// EOM

macro ENL(u)[0.5*(dx(u#1)^2 + dx(u#2)^2 + dx(u#3)^2),
             0.5*(dy(u#1)^2 + dy(u#2)^2 + dy(u#3)^2),
             0.5*(dz(u#1)^2 + dz(u#2)^2 + dz(u#3)^2),
                  dy(u#1)*dz(u#1) + dy(u#2)*dz(u#2) + dy(u#3)*dz(u#3),
                  dx(u#1)*dz(u#1) + dx(u#2)*dz(u#2) + dx(u#3)*dz(u#3),
                  dx(u#1)*dy(u#1) + dx(u#2)*dy(u#2) + dx(u#3)*dy(u#3)]// EOM

macro DENL(u, v)[dx(u#1)*dx(v#1) + dx(u#2)*dx(v#2) + dx(u#3)*dx(v#3),
                 dy(u#1)*dy(v#1) + dy(u#2)*dy(v#2) + dy(u#3)*dy(v#3),
                 dz(u#1)*dz(v#1) + dz(u#2)*dy(v#2) + dz(u#3)*dy(v#3),
                 dy(u#1)*dz(v#1) + dy(u#2)*dz(v#2) + dy(u#3)*dz(v#3)
               + dz(u#1)*dy(v#1) + dz(u#2)*dy(v#2) + dz(u#3)*dy(v#3),
                 dz(u#1)*dx(v#1) + dz(u#2)*dx(v#2) + dz(u#3)*dx(v#3)
               + dx(u#1)*dz(v#1) + dx(u#2)*dz(v#2) + dx(u#3)*dz(v#3),
                 dx(u#1)*dy(v#1) + dx(u#2)*dy(v#2) + dx(u#3)*dy(v#3)
               + dy(u#1)*dx(v#1) + dy(u#2)*dx(v#2) + dy(u#3)*dx(v#3)]// EOM

macro DE(u, v)(EL(v) + DENL(u, v))// EOM

macro E(u)(EL(u) + ENL(u))// EOM

macro D2E(u, v, w)DENL(v, w)// EOM
```

Algorithm 3.8: Nonlinear elasticity kinematic operators computed with FreeFem++.

Obviously using eq. (3.8),

$$\begin{aligned} C_{11} &= 2\varepsilon_{11} + 1 \quad C_{23} = 2\varepsilon_{23} \\ C_{22} &= 2\varepsilon_{22} + 1 \quad C_{31} = 2\varepsilon_{31} \\ C_{33} &= 2\varepsilon_{33} + 1 \quad C_{12} = 2\varepsilon_{12}\,. \end{aligned}$$

Hence, if the eigenvalues of C are $(\lambda_1, \lambda_2, \lambda_3)$[1],

$$\begin{aligned} I_C &= \lambda_1^2 + \lambda_2^2 + \lambda_3^2 \\ II_C &= (\lambda_2\lambda_3)^2 + (\lambda_3\lambda_1)^2 + (\lambda_1\lambda_2)^2 \\ III_C &= (\lambda_1\lambda_2\lambda_3)^2\,. \end{aligned}$$

These are local tensors that must be retrieved at each node associated with the unknowns of the finite element spaces: this is a costly step for assembling each linear system, but since the computations are concurrent, domain decomposition methods allow for significant speedups during assembly of the systems. The approach described so far is based on, first, a linearization of the global problem, followed by a domain decomposition to solve eq. (3.6). It has proven to be successful, for example in [De Roeck 1993; De Roeck, Le Tallec, and Vidrascu 1992]. Another approach introduced by Cai and Keyes [2002] for overlapping Schwarz methods is based on, first, a decomposition of the global problem, followed by local linearizations. The latter is more robust, especially when there are unbalanced nonlinearities in the system. Similar techniques have been developed for nonoverlapping methods, cf. [Klawonn, Lanser, and Rheinbach 2014].

Time-dependent linear elasticity

For unsteady problems, the elastic wave equation involves linear elasticity with variation in time, and reads for $u \in [H_0^1(\Omega)]^3$ and $t \in [0;T]$:

$$\rho\frac{\partial^2 u}{\partial t^2} = c\frac{\partial u}{\partial t} + \nabla\cdot(\mathcal{C} : \varepsilon) + f,$$

where ρ is the density of the material, c is the viscosity, $\mathcal{C}$ is the fourth-order stiffness tensor, and f is a source term. Standard spatial finite element discretization yields the following system:

$$\mathbb{M}\frac{\partial^2 u}{\partial t^2} = \mathbb{C}\frac{\partial u}{\partial t} + \mathbb{K}u + \mathbb{F}\,.$$

$\mathbb{M}$ (resp. $\mathbb{K}$) is usually referred to as the mass (resp. stiffness) matrix.
The Newmark [1959] scheme reads, with initial temporal boundary conditions $(u_0, \dot{u}_0)$:

1. initialize $\ddot{u}_0 = \mathbb{M}^{-1}(\mathbb{C}\dot{u}_0 + \mathbb{K}u_0 + \mathbb{F})$,
2. loop on n, with a time step Δt,
   (a) compute explicitly:

$$\begin{aligned} u_{n+1/2} &= u_n + \Delta t\,\dot{u}_n + (1-2\beta)\frac{\Delta t^2}{2}\ddot{u}_n \\ \dot{u}_{n+1/2} &= \dot{u}_n + (1-\gamma)\Delta t\,\ddot{u}_n\,, \end{aligned}$$

[1] Computed using a dense eigenvalue solver, e.g. `?sterf` from LAPACK.

(b) solve the following linear system,

$$(\mathbb{M} + \gamma\Delta t\mathbb{C} + \beta\Delta t^2\mathbb{K})\ddot{u}_{n+1} = (\mathbb{K}u_{n+1/2} + \mathbb{F}) \,,$$

(c) advance speed and displacement:

$$\begin{aligned} u_{n+1} &= u_{n+1/2} + \beta\,\Delta t^2\ddot{u}_{n+1} \\ \dot{u}_{n+1} &= \dot{u}_{n+1/2} + \gamma\,\Delta t\,\ddot{u}_{n+1} \,. \end{aligned}$$

Values $(\gamma, \beta) \in \mathbb{R}^2$ such that $\frac{1}{2} \leqslant \gamma \leqslant 2\beta$ lead to an unconditionally stable scheme. A common choice for those values is $(\gamma, \beta) = (1/2, 1/4)$. Once again, domain decomposition preconditioners may be used to accelerate the assembly and solution of each linear system.

In the conclusion, page 103, it will be explained why the two previous paragraphs concerning nonlinear and unsteady problems lack numerical experiments: there are currently too many open questions and there is no clear satisfying answer.

3.2 Feel++

Feel++ [Prud'homme et al. 2012] is a domain-specific embedded language (DSEL) written in C++ that is suitable for generalized Galerkin methods. Recently [Chabannes 2013], a lot of effort has been put into efficiently parallelizing the library, so that end-users do not have to deal with message passing, multithreading, or offloading computations to coprocessors. Just as for FreeFem++, some tools and operators needed for building domain decomposition preconditioners will now be explained. However, unlike FreeFem++ which is only equipped with sequential finite element kernels, Feel++ is able to transparently switch from local to distributed operators. While this must be handled with care since the communications are now hidden from an end-user, this provide a very effective way for generating the necessary structures.

Mesh construction, using Gmsh [Geuzaine and Remacle 2009] and `.geo` files[2] either generated on the fly by Feel++ for simple geometries or created by the user beforehand.

```
auto unitCube = loadMesh(_mesh = new Mesh<Simplex<3>>);
auto customGeo = createGMSHMesh(_mesh = new mesh_type,
                                _desc = geo(_filename = "tripod.geo",
                                            _dim = 3, _h = 0.2));
```

Mesh extraction, using markers assigned to each geometric entities such as points, faces, or elements. It must be noted that meshes are automatically partitioned by Feel++, and they store additional information such as so-called ghost elements. It is however possible to create "sequential meshes" that do not keep these. This will be useful for computing unassembled operators in the context of substructuring methods.

```
auto traceCube = createSubmesh(unitCube, boundaryfaces(unitCube)); // distributed mesh
auto localGeo = createSubmesh(customGeo, elements(customGeo),
                              Environment::worldCommSeq()); // local mesh
```

[2] The default file format for describing geometries than can be meshed by Gmsh, see figs. A.1 and A.2 for examples.

Finite element space definition, that can either be parallel—which requires communications and the creation of a global numbering—or purely local to a process.

```
auto U = FunctionSpace<Mesh<Simplex<3>>, // distributed space
                       bases<Lagrange<1, Scalar>>>::New(_mesh = unitCube);
auto W = FunctionSpace<Mesh<Simplex<3>>::trace_mesh_type, // distributed space
                       bases<Lagrange<1, Scalar>>>::New(_mesh = traceCube);
auto V = FunctionSpace<Mesh<Simplex<3>>, // local space
                       bases<Lagrange<2, Scalar>>>::New(_mesh = localGeo,
                             _worldscomm = Environment::worldsCommSeq(1));
```

Finite element space linear interpolation, from a space U to another space W.

```
auto op = opInterpolation(_domainSpace = U, _imageSpace = W);
// collective or local computations, depending on the underlying meshes
```

Variational formulation integration, of both linear and bilinear forms. For eq. (1.5), it would lead to lines of code similar to:

```
auto u = Vh->element();
auto v = Vh->element();
auto l = form1(_test = U);
l = integrate(_range = elements(unitCube), _expr = f * id(v));
auto a = form2(_trial = U, _test = U);
a = integrate(_range = elements(unitCube), _expr = gradt(u) * trans(grad(v)));
a += on(_range = markedfaces(unitCube, "Dirichlet"), _rhs = l, _element = u,
        _expr = cst(0.0));
```

Parallel solution of linear systems, which is not needed by the framework but is explained nonetheless since the interface is quite concise and self-explanatory,

```
a.solve(_rhs = l, _solution = u);
```

The various steps for interfacing the framework with Feel++ will be explained in the following sections. The idea is to use Feel++ capabilities for distributing an initial mesh and get the extra information stored with the mesh to initialize the correct structures of the preconditioners. After that, Feel++ parallel capabilities are not required, in particular, there is no need to assemble the global linear system eq. (1.3). Because it is harder to manipulate—glue or extract—meshes at runtime with Feel++ than with FreeFem++, using the recursion process described page 16 for building overlapping decompositions is not achievable with the former. Thus, only the substructuring preconditioners are supported by this approach.

3.2.1 Preprocessing steps

The first step is to actually generate and distribute a mesh. Then, only the mesh local to the current process is extracted and it is used to build a finite element space.

```
using namespace HPDDM;
auto mesh = loadMesh(_mesh = new Mesh<Simplex<Dim>>);
auto localMesh = createSubmesh(mesh, elements(mesh), Environment::worldCommSeq());
auto VhLocal = FunctionSpace<Mesh<Simplex<Dim>>,
                             bases<Lagrange<Order, Type>>>::New(_mesh = localMesh,
                                   _worldscomm = Environment::worldsCommSeq(1));
```

Algorithm 3.9: Initialization of the local finite element space.

3.2.2 Transfer operators

Using the structure for distributed meshes of Feel++ , it is quite easy to build the transfer operators by looping through subdomains that share at least one geometric entity with the local mesh.

```
std::vector<std::vector<int>*> map(mesh->faceNeighborSubdomains().size()); // #O_i
for(int i = 0; i < map.size(); ++i) { // foreach neighbor
   std::set<rank_type>::iterator it = mesh->faceNeighborSubdomains().begin();
   std::advance(it, i);
   auto trace = createSubmesh(mesh, interprocessfaces(mesh, *it),
                        Environment::worldCommSeq()); // create the interface
   auto Xh = FunctionSpace<typename Mesh<Simplex<Dim>>::trace_mesh_type,
                           bases<Lagrange<Order, Type>>>::New(_mesh = trace,
                                 _worldscomm = Environment::worldsCommSeq(1));
   // define the corresponding space
   auto l = Xh->element();
   std::iota(l.begin(), l.end(), 1.0); // initialize the function on the interface

   auto op = opInterpolation(_domainSpace = VhLocal,
                             _imageSpace = Xh,
                             _backend = backend(_worldcomm = Environment::worldCommSeq()),
                                                _ddmethod = true);
   auto opT = op->adjoint(MATRIX_TRANSPOSE_UNASSEMBLED);
   // no need for an explicit assembly
   uLocal = (*opT)(l); // interpolate the function on the subd.
   // nonzero values of this function are on the interface
   map[i] = new std::vector<int>(Xh->nDof());
   for(int j = 0; j < VhLocal->nDof(); ++j)
      if(std::round(uLocal[j]) != 0)
         (*map[i])[std::round(uLocal[j]) - 1] = j;
         // this is the numbering in the local subd.
         // a renumbering in the local trace is needed afterwards
}
```

Algorithm 3.10: Numbering of the interfaces with the neighboring subdomains.

The structure `mapping` can now be renumbered to provide the same connectivity information as in algorithm 2.11 that are needed for the transfer operators on each subdomain $i \in [\![1; N]\!]$: $\left\{B^{(i)}B^{(j)^T}\right\}_{j\in\mathcal{O}_i}$ for the BDD method or $\left\{\underline{B}^{(i)}\underline{B}^{(j)^T}\right\}_{j\in\mathcal{O}_i}$ for the FETI method.

```
std::set<int> unique;
// remove duplicates by using a set
for(std::vector<int>* pt : map)
   for(int& i : *pt)
      unique.insert(i);
interface.insert(interface.begin(), unique.cbegin(), unique.cend());
std::unordered_map<int, int> mapping;
mapping.reserve(interface.size());
int j = 0;
for(const int& i : interface) // create a permutation from subd. to trace
   mapping[i] = j++;
for(std::vector<int>* pt : map)
   for(int& i : *pt) // permute the numbering
      i = mapping[i];
```

Algorithm 3.11: Renumbering of the transfer operators.

3.2.3 Retrieving a local matrix

Because Feel++ uses PETSc internally, it is necessary to access the raw representation of the sparse matrices and vectors after assembly of the local linear systems, and copy those inside simpler structures. This is detailed in the following algorithm.

```
boost::shared_ptr<Backend<double>> ptr_backend = Backend<double>::build(BACKEND_PETSC,
    Environment::worldCommSeq());
Backend<double>::sparse_matrix_ptrtype A = ptr_backend->newMatrix(VhLocal, VhLocal);
Backend<double>::vector_ptrtype f = ptr_backend->newVector(VhLocal);
auto a = form2(_trial = VhLocal, _test = VhLocal, _matrix = A);
a = integrate(_range = elements(localMesh), _expr = gradt(uLocal) * trans(grad(vLocal)));
auto l = form1(_test = VhLocal, _vector = f);
l = integrate(_range = elements(localMesh), _expr = id(vLocal));
if(nelements(markedfaces(localMesh, "Dirichlet")) > 0)
   a += on(_range = markedfaces(localMesh, "Dirichlet"), _rhs = l, _element = uLocal,
           _expr = cst(0.0));
A->close(); // assemble the local linear system
Mat PetscA = static_cast<MatrixPetsc<double>*>(&*A)->mat(); // PETSc matrix
Vec PetscF = static_cast<VectorPetsc<double>*>(&*f)->vec(); // PETSc vector
PetscInt n;
const PetscInt* ia;
const PetscInt* ja;
PetscScalar* array;
PetscBool done;
MatGetRowIJ(PetscA, 0, PETSC_FALSE, PETSC_FALSE, &n, &ia, &ja, &done);
MatSeqAIJGetArray(PetscA, &array);
// retrieve (row_ptr, col, val), cf. section 2.1.2
int nnz = ia[n];
double* c = new double[nnz];
int* ic = new int[n + 1];
int* jc = new int[nnz];
std::copy(array, array + nnz, c);
std::copy(ia, ia + n + 1, ic);
std::copy(ja, ja + nnz, jc);
MatrixCSR<double>* pt = new MatrixCSR(n, n, nnz, c, ic, jc, 0); // internal structure
MatSeqAIJRestoreArray(PetscA, &array);
MatRestoreRowIJ(PetscA, 0, PETSC_FALSE, PETSC_FALSE, &n, &ia, &ja, &done);
```

Algorithm 3.12: Assembling and retrieving the local matrix and right-hand side.

3.2.4 Calling the solver

First, the solver must be instantiated and a choice has to be made whether the FETI or the BDD method is to be used. Just as in algorithm 2.14, `SUBDOMAIN` is a local solver and `COARSEOPERATOR` is a distributed solver for a potential coarse operator.

```
#ifdef FETI
Feti<SUBDOMAIN, COARSEOPERATOR, FetiPrcdtnr::DIRICHLET, 'S', double> K;
#else
Bdd<SUBDOMAIN, COARSEOPERATOR, 'S', double> K;
#endif
```

Algorithm 3.13: Instantiation of the substructuring preconditioner.

Then, it is necessary to compute the rigid body modes that will be needed by an eventual coarse operator in case no generalized eigenvalue problem is solved locally—in which case the kernel of each local Schur complement is supposed to be retrieved algebraically.

```
bool isFloating = (nelements(markedfaces(localMesh, "Dirichlet")) == 0);
if(isFloating) {
   auto rbm = VhLocal->element();
   rbm = vf::project(VhLocal, elements(localMesh), cst(1.0));
   unsigned short nbRbm = 1;
   ev = new double*[nbRbm];
   *ev = new double[nbRbm * VhLocal->nDof()];
   for(unsigned short i = 0; i < nbRbm; ++i) {
      ev[i] = *ev + i * VhLocal->nDof();
      std::copy(rbm.begin(), rbm.end(), ev[i]);
   }
   K.setVectors(ev);
   K.super::super::initialize(nbRbm);
}
else
   K.super::super::initialize(0);
```

Algorithm 3.14: Building the local rigid body modes.

It is now time to factorize the local matrices $\left\{\mathring{A}_i\right\}_{i=1}^{N}$ as well as assemble, redistribute, and factorize the coarse operator.

```
K.renumber(interface, b);
K.callNumfactPreconditioner();
std::string scaling = soption("scaling");
K.buildScaling(scaling[0]);
std::vector<unsigned short> parm(5);
parm[Parameter::P] = ioption("p");
parm[Parameter::TOPOLOGY] = ioption("topology");
parm[Parameter::DISTRIBUTION] = NON_DISTRIBUTED;
parm[Parameter::STRATEGY] = ioption("strategy");
parm[Parameter::NU] = K.getLocal();
K.callNumfact();
K.buildTwo(parm);
```

Algorithm 3.15: Factorization of the local linear systems and of the coarse operator.

Eventually, the preconditioned CG may now be called to solve the substructured problem.

```
unsigned short iter = ioption("it");
double eps = doption("eps");
IterativeMethod::PCG(K, x, b, iter, eps, MPI_COMM_WORLD, Environment::isMasterRank());
K.originalNumbering(interface, x);
```

Algorithm 3.16: Calling the preconditioned CG.

Chapter 4

Improving scalability

The performance of the framework presented in chapter 2 is further increased in this chapter. In particular, some scalability issues when running at very large-scale are tackled. Some of these results, mainly in section 4.1, were initially published in [Jolivet et al. 2013]. Sections 4.3 and 4.4 gather techniques that are particularly efficient when the cost of synchronization is high, for example when using accelerators such as GPU. Readers only interested in numerical experiments should skip to chapter 5.

On propose dans ce chapitre des moyens d'améliorer les performances de la librairie présentée chapitre 2. Des problèmes de passage à l'échelle pour de très grandes simulations sont notamment réglés. Certains de ces résultats, en particulier ceux de la section 4.1, ont été initialement publiés dans [Jolivet et al. 2013]. Les sections 4.3 et 4.4 rassemblent des techniques particulièrement intéressantes lorsque le coût des synchronisations devient trop important, par exemple lorsque l'on utilise des accélérateurs comme les GPU. Le lecteur uniquement intéressé par les résultats numériques est invité à se rendre directement au chapitre 5.

Contents

4.1 Distribution of the coarse operator 81

4.2 Subdomain-level parallelism 86

4.3 Separation of tasks 86

4.4 Synchronization-avoiding Krylov solver 87

Ensuring proper scaling of parallel code is of paramount importance if the code is to be used on large architectures. One particular key feature of implementations of domain decomposition preconditioners or multigrid methods is the handling of coarse operators. That is because a naive approach would lead to all-to-one and one-to-all types of communications. This chapter proposes various ways to improve and leverage the communication pattern necessary to perform coarse grid correction.

4.1 Distribution of the coarse operator

Some of the ideas behind the assembly and the use of the coarse operator described section 2.2.2 are not very new to the field of domain decomposition methods since they were already introduced in some prior works, cf. [Bhardwaj, Day, et al. 2000; Bhardwaj, Pierson, et al. 2002; Roux and Farhat 1998]. The problem with these static implementations though, is that they cannot scale to large numbers of subdomains because having only one process to handle the coarse operator can quickly become an issue. In more recent work, people

have tried to distribute the coarse operator and they were able to significantly speedup their code.

1. Klawonn and Rheinbach [2010] use PETSc to redistribute the coarse operator, and then call BoomerAMG [Henson and U. M. Yang 2002] to solve the coarse problem in parallel. A similar approach is described by Kozubek et al. [2013].

2. Badia, Martín, and Príncipe [2013] use one MPI process with multiple threads to handle the coarse operator.

These approaches can provide great results but lack genericity. Another technique presented in [Jolivet et al. 2013, 2014b] can be used to assemble the coarse operator on an arbitrary number of master processes, without the need to redistribute the Galerkin matrix. The idea is to use only a "small" group of processes that will be in charge of factorizing the coarse operator and that will afterwards be called for computing solutions of systems involving E^{-1} using a distributed sparse direct solver. This is inspired by the famous master/slave approach. The following notations will be thoroughly used in the following sections. For the sake of completeness, their type in the actual implementation are provided:

- `unsigned short` P: the number of masters, chosen at runtime by the user,
- `MPI_Comm masterComm`: a communicator between all masters, set to `MPI_COMM_NULL` on slaves, on which will be instantiated the distributed solver,
- `MPI_Comm splitComm`: a communicator between a master and its slaves in which the rank of the master is always 0, and the ranks of the slaves follow the same order as in the default communicator.

A representation of such communicators has already been given fig. 2.6 and will also be depicted fig. 4.3. Prior to factorizing E, the first step is to assemble the operator in a distributed matrix on the masters. Each master will have to assemble all the values of its slaves. It is assumed that the distributed matrix is stored in plain distributed CSR or COO format. Few codes could scale to important numbers of subdomains without a sparse data structure for the coarse operator, for example in [Grigori, Stompor, and Szydlarski 2012], the Galerkin matrix E is stored in a dense array and replicated on each MPI process: this is only possible because there is just one deflation vector per process, so its size is quite small compared to the one assembled and factorized in the context of domain decomposition preconditioners. Since the structure of the matrix is sparse, for each nonzero value, the absolute row and column indices of the given value in the matrix E must be known. For a process $i \in [\![1; N]\!]$, the global row indices of all the blocks $\{E_{i,j}\}_{j \in \overline{\mathcal{O}}_i}$ range from r_i to $r_i + \nu_i$, where $r_i = \sum_{j=1}^{i-1} \gamma_j$, and the global column indices of all the blocks $\{E_{j,i}\}_{j \in \overline{\mathcal{O}}_i}$ range from r_i to $r_i + \nu_i$. The simplest approach would then be to:

1. call `MPI_Allgather`(γ_i) on the global communicator to be able to compute the cumulative sums $\{r_j\}_{j \in \overline{\mathcal{O}}_i}$ and to allocate the buffers $\mathcal{S}_j^{(i)}$ and $\mathcal{R}_j^{(i)}$ defined section 2.2.2 for all $j \in \mathcal{O}_i$,

2. assemble locally $\{E_{i,j}\}_{j \in \overline{\mathcal{O}}_i}$ as previously and store the values in sparse format, for example using CSR storage into $(row\_ptr_i, col_i, val_i)$,

3. call `MPI_Gatherv`(row_ptr_i), `MPI_Gatherv`(col_i), and `MPI_Gatherv`(val_i) on `splitComm` with rank 0 as root.

The number of nonzero values for a process i is:

$$\texttt{size}(val_i) = \underbrace{\gamma_i \times \gamma_i}_{\text{diagonal values } E_{ii}} + \underbrace{\gamma_i \times \sum_{j \in \mathcal{O}_i} \gamma_j}_{\text{off-diagonal values } \{E_{ij}\}_{j \in \mathcal{O}_i}}, \tag{4.1}$$

meaning that prior to the three `MPI_Gatherv` in step 3, a call to `MPI_Gatherv`($\mathcal{O}_i$) on `splitComm` with rank 0 as root must be made to allocate the right arrays for the distributed sparse data structure on each master. While this approach is somehow natural for assembling the distributed matrix on the masters (because of the ordering of the rank of the slaves, calling `MPI_Gatherv` is similar to concatenating all local chunks of E) it implies a lot of unnecessary communications. In particular, why should slaves send to masters the global row and column indices ? Indeed, at the end of assembly, only the masters have access to the distributed coarse operator, so it is their responsibility to compute the indices. The slaves should not have to store or compute anything related to the distributed format. The following approach has the advantage of transferring only what is needed from one slave i to its master: the array of scalar val_i. The indices will be computed after reception by each master, meaning that the memory overhead on the slaves is null: no integer is allocated for storing any index. An improved workflow is:

1. perform a neighborhood collective operation `MPI_Ineighbor_alltoall`(ν_i)[1] on the communicator to which the distributed graph topology information of the connectivity between subdomains is attached (`MPI_Dist_graph_create_adjacent`). Then allocate accordingly the buffers $\mathcal{S}_j^{(i)}$ and $\mathcal{R}_j^{(i)}$ for all $j \in \mathcal{O}_i$,

2. call `MPI_Gather`($\begin{bmatrix}\gamma_i & \#\mathcal{O}_i\end{bmatrix}$) (array of 2 integers) on `splitComm` with rank 0 as root so that masters can preallocate the distributed sparse data structure $(row\_ptr_i, col_i, val_i)$,

3. first, prepend to the beginning of a message the values of $\mathcal{O}_i$. Then, assemble locally $\{E_{ij}\}_{j \in \overline{\mathcal{O}}_i}$ as previously and send the values to the master, i.e. the final size of the message is $\#\mathcal{O}_i +$ (4.1).

Additionally, the masters must concatenate all γ_i gathered in step 2, using `MPI_Allgatherv` on `commMaster` to be able to compute all cumulative sums r_i, for all $i \in$ `splitComm`. This call is equivalent to the `MPI_Allgather` in step 1 of the initial algorithm, but this time it does not involve any slave. When a master receives a message from a slave i, it knows that the global row index ranges from r_i to $r_i + \gamma_i$, and because the first values of the received message are a copy of $\mathcal{O}_i$, it can compute the correct global column indices for the neighboring subdomains of this slave. The complete algorithm for assembling the coarse operator is summarized in algorithms 4.1 (construction of all local blocks of E) and 4.2 (distributed assembly on the masters).

[1] New to the MPI-3 standard, www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

Algorithm 4.1: Schematic construction of E_{ij}, $\forall i \in [\![1; N]\!]$ and $\forall j \in \overline{\mathcal{O}}_i$.

```
MPI_Ineighbor_alltoall(γ_i)
MPI_Gather([γ_i  #O_i], splitComm, 0)
compute(T_i)                                   // T_i = A_ii W_i or D_i S_i^⋆ D_i W_i
foreach j ∈ O_i do
    MPI_Isend(S_j^(i), j)
    MPI_Irecv(R_j^(i), j, default communicator, rq[j])
compute(E_ii)                                  // diagonal block
foreach j ∈ O_i do
    MPI_Waitany(rq, &index)
    compute(E_i index)                         // off-diagonal block
```

Algorithm 4.2: Schematic assembly of E on master processes.

```
1  buildComm(P, splitComm, masterComm)
2  if masterComm != MPI_COMM_NULL then                                  // master
3      MPI_Allgatherv(ν_i)          // now receive all messages from the slaves
4      for j ← 1 to MPI_Comm_size(splitComm) do
5          MPI_Irecv(msgFromSlave[j], j − 1, splitComm, rq[j])
6      assemble(E_ii)
7      for k ← 1 to #O_i do
8          assemble(E_ik)
       // blocks local to the masters have been assembled
9      for j ← 1 to MPI_Comm_size(splitComm) do
10         MPI_Waitany(rq, &index)
11         assemble(E_index index)
12         for k ← 1 to #O_index do
13             assemble(E_index msgFromSlave[index][k])
       // blocks from the slaves have been assembled
14     numericalFactorization(E)
15 else                                                                 // slave
16     msgToMaster = O_i
17     concatenate(msgToMaster, E_ii)
18     foreach k ∈ O_i do
19         concatenate(msgToMaster, E_ik)
20     MPI_Isend(msgToMaster, 0, splitComm)
       // send O_i and the local values of E computed in algorithm 4.1
```

The procedure `compute` in algorithm 4.1 is in charge of returning a dense array of values corresponding to a local block of E given in argument, while the procedure `assemble` in algorithm 4.2 is in charge of computing the global row and column indices of the values of a block of E given in argument and store them in the distributed sparse matrix data structure.

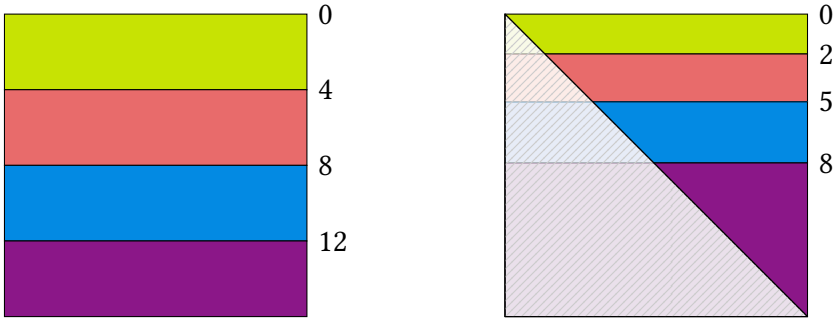
Electing the masters

In the previous paragraph, the masters are defined in a rather abstract way, as the processes that have a rank equal to 0 in `splitComm`. Two ways to define the aforementioned MPI communicator will now be explained. The first is the natural distribution: processes are spread uniformly and contiguously into P groups, the masters are of rank $i \cdot N/P$, for all $i \in [\![0; P-1]\!]$. The second distribution is a little more advanced and better suited for assembling symmetric coarse operators. In that case, one only needs to assemble the upper part of the distributed sparse structure, so that only the following blocks are computed and assembled:

$$E_{ij},\ \forall i \in [\![1; N]\!] \text{ and } \forall j \in \overline{\mathcal{O}}_i\ :\ j \geqslant i\,.$$

Moreover, only the upper parts of the dense diagonal blocks $\{E_{ii}\}_{i=1}^{N}$ are needed. To ensure load balancing between masters, processes are now spread contiguously but nonuniformly, with masters of rank p_i, where p_i is defined by the following sequence to ensure heuristically that the number of values within each quadrilateral in fig. 4.3 is the same:

$$\begin{aligned} p_0 &= 0 \\ \forall i \in [\![1; P-1]\!],\ p_i &= \left\lfloor N - \sqrt{(p_{i-1} - N)^2 - N^2/P} + 0.5 \right\rfloor . \end{aligned}$$



(a) Uniform distribution **(b)** Nonuniform distribution

Fig 4.3: Sixteen subdomains split among four masters. Each color represents a different `splitComm`, each number represents the rank of the master (in the default communicator) of a `splitComm`. On the right figure, the number of values per `splitComm` is roughly the same if the lower triangular part of the matrix is dropped (symmetric Galerkin matrix).

Coarse correction

The workflow explained page 49 and depicted fig. 2.7 for computing a coarse correction with a single process can easily be extended to the case of multiple master processes. This time, the preprocessing step will consist of the parallel factorization of the sparse matrix E which is distributed on the MPI communicator made of P processes `masterComm`. A coarse correction can then by broken down into four elementary operations:

1. compute locally $v^{(i)} = W_i^T u^{(i)}$ and gather those values inside v on process 0 of `splitComm`,
2. compute $x = E^{-1}v$ on `masterComm` using a distributed right-hand side and solution,
3. scatter x from process 0 of `splitComm` inside $x^{(i)}$ on each process and compute locally $y^{(i)} = W_i x^{(i)}$,
4. perform the reduction in the same way as in step 4 page 50.

A nice property of this approach is that there is no global communication involving the default communicator made of N processes when applying a two-level preconditioner. Instead, global communications are made on all P communicators `splitComm`.

4.2 Subdomain-level parallelism

So far, the only parallel programming model detailed in the framework is message passing. Another model that can be beneficial in the context of domain decomposition methods is shared memory multiprocessing programming. This is useful for speeding up tasks that must be performed concurrently on each subdomain, such as factorizing local systems of linear equations or solving generalized eigenvalue problems. Since the framework depends on BLAS, LAPACK, and a direct solver, most of its computational intensive kernels are already multithreaded as long as those third-party dependencies are. This is usually the case when using vendor implementations of these libraries, such as IBM *Engineering and Scientific Subroutine Library*. In chapter 5, the numerical experiments are performed using the MKL and MUMPS or PARDISO as direct solvers, so that all numerical algebraic operations in the framework are multithreaded. It is also possible to parallelize the assembly of the coarse operator E. In particular, threads may be forked for the loop lines 9–13 in algorithm 4.2. This should decrease the time needed for assembly, especially if the MPI implementation supports a level of thread support equal to `MPI_THREAD_MULTIPLE`. An important point to take into account when running at large-scale is to ensure the right placement of MPI processes. In the numerical experiments of the following section, each subdomain is bound to a socket, and sequential MPI processes are on adjacent processor cores[2].

4.3 Separation of tasks

In this section, it is assumed that a one-level preconditioner is enriched with a coarse operator additively. Such a preconditioner is given eq. (1.17). Substructuring preconditioners presented in section 1.2 do not satisfy this assumption, however, it is possible to modify them so that only an additive coarse correction is needed, cf. [Badia, Martín, and Principe 2013]. In that case, one can write $P^{-1} = \Xi^{-1} + M^{-1}$, where M^{-1} is a one-level preconditioner and Ξ^{-1} is a coarser second level preconditioner. With overlapping Schwarz methods, M^{-1} can be chosen as $\sum_{i=1}^{N} R_i^T D_i {A_{ii}}^{-1} R_i$ while $\Xi^{-1} = Z^T E^{-1} Z$. Using the work of section 4.1, while factorizing or solving a linear system involving the Galerkin matrix E, the P processes of `masterComm` are calling a distributed solver, while the $N - P$ slaves are idle. Moreover, master processes have no less than two systems to solve: at least one from the one-level preconditioner M^{-1}, and another one concerning E^{-1}. To avoid this load imbalance, one idea is to separate fine-grained local tasks of the first level from coarse-grained global tasks of the second level. This can be done by excluding the masters from the initial domain partitioning. That way,

- factorizing E and the local matrices $\{A_{ii}\}_{i=1}^{N}$ is done concurrently during the setup of the preconditioner,
- applying the one-level preconditioner M^{-1} only requires local computations from slaves and point-to-point communications between slaves only,

[2] With OpenMPI [Gabriel et al. 2004], this can be achieved using the following command-line arguments: `--bind-to-socket --bycore`. With MPICH [Balaji et al. 2014], the arguments are: `-bind-to socket -map-by hwthread`.

- computing a coarse correction using Ξ^{-1} still requires communication from slaves to master and vice versa, but only computations from masters.

To avoid the synchronizations needed by the gathering and scattering of values by master processes steps 1 and 3 of the workflow explained page 85, it is possible to use asynchronous collective operations. When P^{-1} has to be applied, the first step slaves must do is compute local $v^{(i)}$ as in step 1 of the previous algorithm, and then start asynchronous communications for both sending and receiving the coarse correction from their master. Meanwhile, masters gather values received from their slaves, compute the coarse correction using the distributed solver, and scatter the solution vector back to their slaves. Asynchronously, slaves apply the one-level preconditioner M^{-1} and afterwards, wait for the completion of the asynchronous collective operations. This leads to highly scalable preconditioners. In fact, if applying M^{-1} takes long enough, computing the coarse correction with Ξ^{-1} can be fully overlapped. Similar work has recently been conducted for additive variants of algebraic multigrid (AMG) preconditioners, cf. [Vassilevski and U. M. Yang 2014].

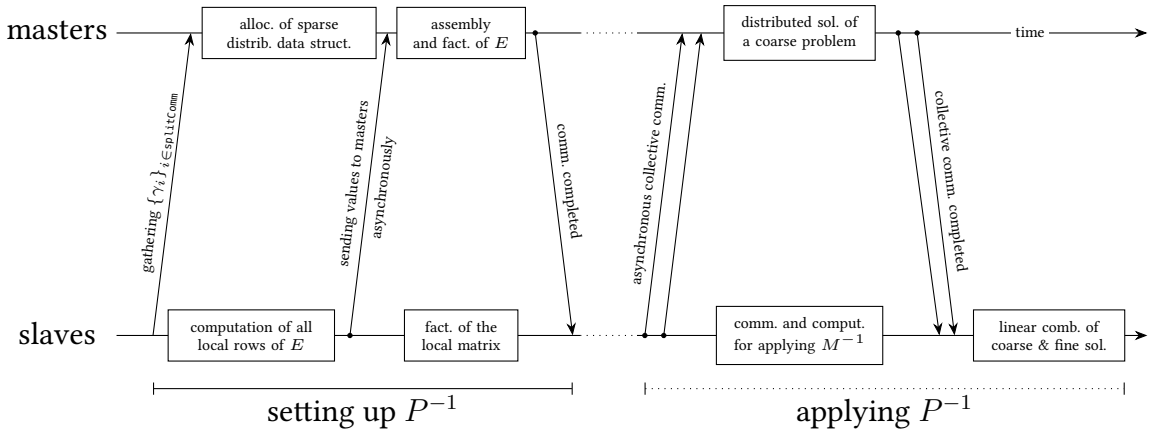


Fig 4.4: Time diagram showing various tasks performed by a two-level method.

4.4 Synchronization-avoiding Krylov solver

The communication cost of an algorithm often dominates its arithmetic cost, and technological trends indicate this cost gap will increase. In the past [Chronopoulos and Gear 1989; de Sturler and Van der Vorst 1995], people have been trying to reduce the communication overhead introduced by inner products in iterative methods—which are limiting their scalability [Bhatele et al. 2011]—at the cost of performing some additional computations. Another approach consists in avoiding unnecessary synchronization by using nonblocking collective operations which provide the ability to overlap communication with computation [Hoefler, Gottschling, et al. 2007; Hoefler, Lumsdaine, and Rehm 2007]. A new variant of the GMRES was recently introduced by Ghysels, Ashby, et al. [2013]. It decreases the number of inner products per iteration of the computational loop from two to one[3], as detailed in algorithm 4.5, as well as facilitates communication/computation overlap using nonblocking collective operations.

[3] In the original paper, the authors only investigate the case of the unpreconditioned GMRES.

Fig 4.5: Computational loop of the p^1-GMRES.

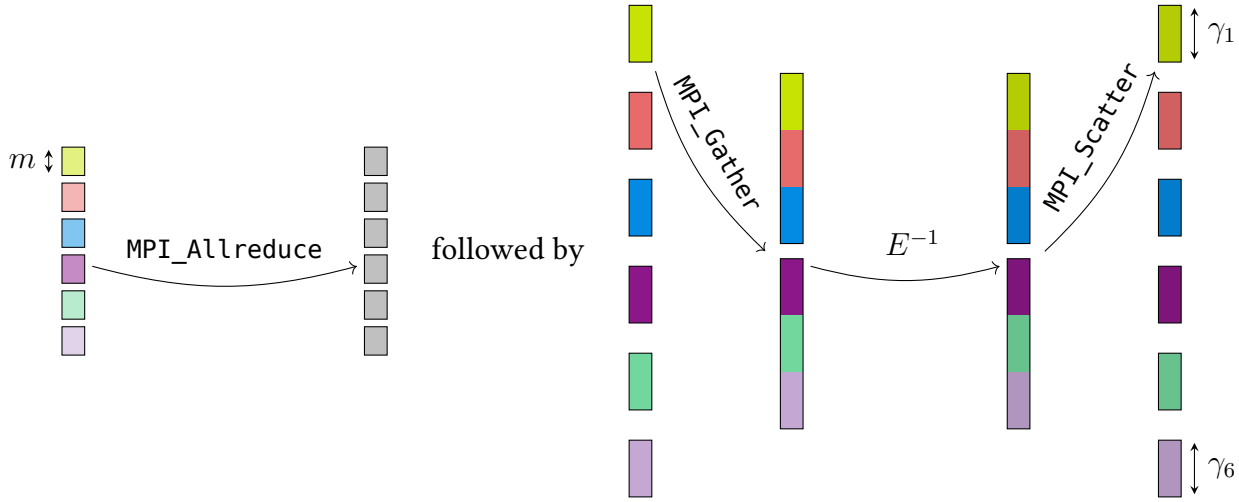
```
1  for i ← 0 to m do
2      w ← P^{-1} A v_i
3      if i > 1 then
4          v_{i-1} ← v_{i-1} / h_{i-1,i-2}
5          z_i ← z_i / h_{i-1,i-2}
6          w ← w / h_{i-1,i-2}
7          h_{i-1,i-1} ← h_{i-1,i-1} / h²_{i-1,i-2}
8          h_{j-1,i-1} ← h_{j-1,i-1} / h²_{i-1,i-2},  j = 0, ..., i − 2
9      z_{i+1} ← w − Σ_{j=0}^{i-1} h_{j,i-1} z_{j+1}
10     if i > 0 then
11         v_i ← z_i − Σ_{j=0}^{i-1} h_{j,i-1} v_j
12         h_{i,i-1} ← ||v_i||_2
13     h_{j,i} ← ⟨z_{i+1}, v_j⟩,  j = 0, ..., i
```

The inner products lines 12 and 13 may be computed together, so that there is effectively only one global all-to-all reduction per iteration. Moreover, the results are not needed until line 4 for the next iteration, thus, the communications involved by the reduction can be overlapped with the computations of the preconditioner-sparse matrix-vector product line 2. As exhibited eqs. (1.12), (1.29) and (1.38), sparse matrix-vector products only require asynchronous point-to-point communications. When preconditioning with simple one-level methods, cf. section 1.1.2 and eq. (1.31), then the same remark holds when applying such preconditioners to the left. However, a two-level method implies a more complex communication pattern, cf. section 2.2.2 and fig. 2.7. It is possible to use this pattern to fuse some communications as explained next. The idea is to reduce values while computing coarse corrections. In the rest of this paragraph, it will be assumed that the symbol $\bullet$ represents a binary operation that can be interpreted as a `MPI_Op` for a call to `MPI_Allreduce` for performing an in-place reduction of m values stored locally in an array h. In the context of the p^1-GMRES, $\bullet$ = `MPI_SUM` and h represents the latest column of the Hessenberg matrix produced by the Arnoldi recurrence—line 13 of algorithm 4.5—and the previous subdiagonal entry—line 12—that both need to be orthonormalized. Then, instead of performing this plain all-to-all reduction followed by a coarse correction, both operations can be executed at the same time. For that, the workflow presented in section 4.1 has to be rewritten. A fused coarse correction–reduction can then by broken down into eight elementary steps:

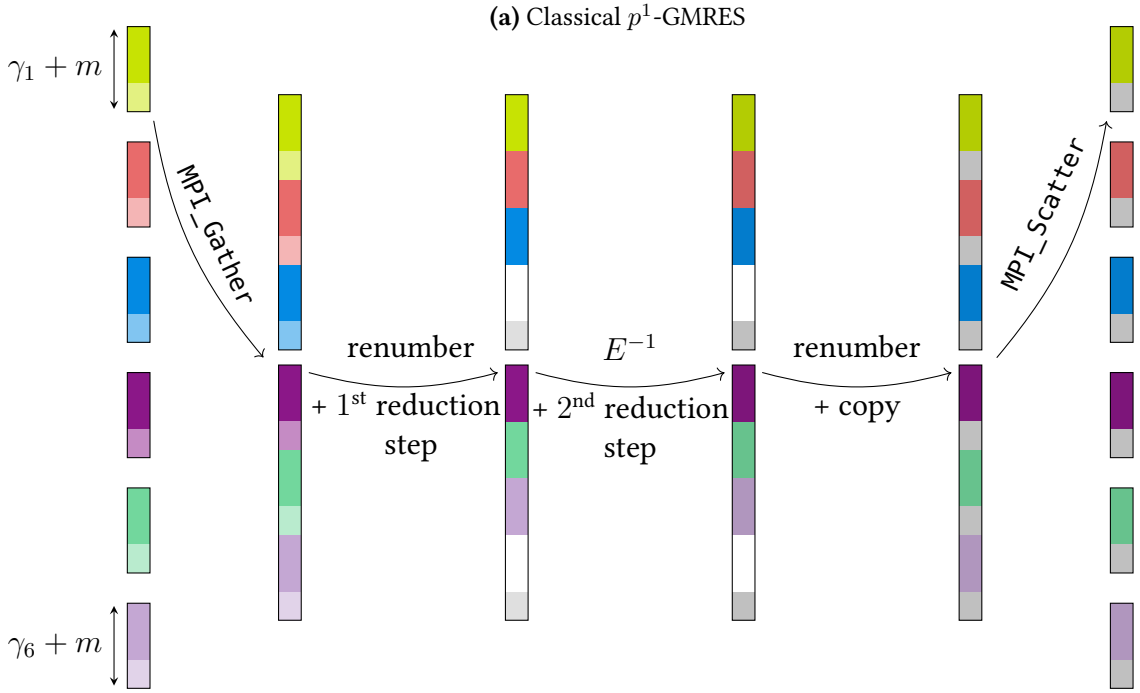
1. compute locally $v^{(i)} = W_i^T u^{(i)}$, append to this vector the m elements of h, and gather all those values inside v on process 0 of `splitComm`,
2. rearrange on each master process v which is now of size $m \times \text{size}(\texttt{splitComm}) + \sum_{i \in \texttt{splitComm}} \gamma_i$ so that the right-hand side is stored contiguously while the first step of the reduction is performed for all the extra $m \times \text{size}(\texttt{splitComm})$ values received,
3. perform the second step of the reduction which is a call to `MPI_Allreduce` on the communicator made of P processes `masterComm`,
4. compute $x = E^{-1} v$ on `masterComm` using a distributed right-hand side and solution,
5. rearrange on each master process x and duplicate the reduced values of h,
6. scatter x from process 0 of `splitComm` inside $x^{(i)}$ on each process and compute locally $y^{(i)} = W_i x^{(i)}$,

7. perform the reduction in the same way as in step 4 page 50,
8. retrieve the reduced values of h in-place, i.e. at the end of $y^{(i)}$.

A graphical explanation of this workflow is given fig. 4.6b.



(a) Classical p^1-GMRES



(b) Fused p^1-GMRES

Fig 4.6: Two possible workflows for the p^1-GMRES using six subdomains and two masters. The right-hand side and solution vectors are kept distributed on master processes.

With this fused algorithm, there is not a single communication on the global communicator. The only communications needed are between masters and between one master and its slaves, putting aside point-to-point communications needed by one-level methods. In conjunction with the separation of tasks described in the previous section for the additive correction, steps 2–5 are executed on master processes while slaves are in charge of one-level preconditioning if the collective operations which gather and scatter vectors can be replaced by nonblocking collective operations. For symmetric problems in substructuring methods, the same approach may be used for the preconditioned CG [Ghysels and Vanroose 2013].

Chapter 5

Numerical experiments

EXTENSIVE experiments will now be carried out using the framework previously described in chapter 2 and further improved in chapter 4. These experiments assess its ability to solve both hard and large problems. In sections 5.2 and 5.3, a comparison with other state of the art linear solvers is drawn.

DES essais numériques extensifs vont être présentés en utilisant la librairie décrite précédemment dans le chapitre 2 et retravaillée dans le chapitre 4. Ces essais démontrent sa capacité à résoudre des problèmes complexes de grande taille. Dans les sections 5.2 et 5.3, elle sera comparée à d'autres solveurs linéaires de pointe.

Contents

5.1 Test bed **91**

5.2 Comparison with multigrid solvers **98**

5.3 Comparison with direct solvers **99**

They are many ways to evaluate the performances of a parallel code. In particular, two common metrics will be studied.

1. Strong scaling gives an insight on how a code performs when the number of processing units is increased for solving a fixed size global problem. In a finite element framework, this can be computed by using a global mesh with a fixed number of elements, and just increasing the number of subdomains for partitioning the mesh.

2. Weak scaling shows how a code behaves when the number of processing units increases, while maintaining local problems with constant sizes. In a finite element framework, this is done by using a global mesh that is refined locally on each subdomain while increasing the number of subdomains for partitioning the mesh. That way, the local numbers of elements remain constant.

Throughout this chapter, only the timings relative to the construction and the use of the preconditioners are considered. In particular, the time spent in the finite element backend is not evaluated.

5.1 Test bed

Results concerning overlapping preconditioners as introduced section 1.3 were obtained on Curie, a Tier-0 system for PRACE[1], with a peak performance of 1.7 PFLOP/s. They have been first published in the article [Jolivet et al. 2013] nominated for the best paper award at SC13[2]

[1] Partnership for Advanced Computing in Europe. URL: http://www.prace-ri.eu/.

[2] Among five other papers out of 90 accepted papers out of 457 submissions.

and were also disseminated in PRACE Annual Report 2013 [Oorsprong et al. 2014, pp. 22–23] as a "success story". Curie is composed of 5 040 nodes made of two eight-core Intel Sandy Bridge processors clocked at 2.7 GHz. Its interconnect is an InfiniBand QDR full fat tree [Leiserson 1985] and the MPI implementation was BullxMPI version 1.1.16.5. Intel compilers and *Math Kernel Library* in their version 13.1.0.146 were used for all binaries and shared libraries, and as the linear algebra backend for both dense and sparse computations in the framework. Finite element matrices are obtained with *FreeFem++*. The speedup and efficiency are displayed in terms of number of MPI processes. In these experiments, each MPI process is assigned a single subdomain and two OpenMP threads following the bindings proposed section 4.2. Because the preconditioner is not symmetric, the underlying iterative method is the GMRES, which is stopped when a relative 10^{-6} decrease of the initial residual is reached.

First, the system of linear elasticity with highly heterogeneous elastic moduli is solved with a minimal geometric overlap of one mesh element in dD ($d = 2$ or 3). It can be derived from eq. (3.1) when assuming that there is a relationship between the stress tensor and the strain tensor such that $\Sigma = \mathcal{C} : \varepsilon$, where $\mathcal{C}$ is fourth-order tensor defined eq. (3.7), and using a linear approximation. Then, the system consists in finding $u \in [H_0^1(\Omega)]^d$ such that:

$$-\nabla \cdot ((I + \nabla u)\,\Sigma) = -\nabla \cdot ((I + \nabla u)\,\mathcal{C} : (\varepsilon_{\mathrm{L}} + \varepsilon_{\mathrm{NL}})) \approx -\nabla \cdot (\mathcal{C} : \varepsilon_{\mathrm{L}}) = f\,.$$

After using Green's formula, its variational formulation is, for all test functions $v \in [H_0^1(\Omega)]^d$:

$$\begin{aligned} a(u,v) &= \int_\Omega \frac{E\nu}{(1+\nu)(1-2\nu)} \nabla \cdot u \nabla \cdot v + \frac{E}{1+\nu} \varepsilon_{\mathrm{L}}(u) : \varepsilon_{\mathrm{L}}(v) \\ l(v) &= \int_\Omega f \cdot v + \int_{\partial\Omega} g \cdot v\,, \end{aligned} \tag{5.1}$$

where:

- Young's modulus E and Poisson's ratio ν vary between two sets of values, $(E_1, \nu_1) = (2 \cdot 10^{11}, 0.25)$, and $(E_2, \nu_2) = (10^7, 0.45)$,
- ε_{L} is the linearized strain tensor introduced eq. (3.2), f are the body forces (in this case, only the gravity), and g are the surface force (in this case, a vertical loading is imposed on some parts of the geometries).

Poisson's ratio being relatively far from the incompressible limit of 0.5, it is not necessary to switch to a mixed finite element formulation since there is no locking effect as described by Babuška and Suri [1992]. Such an equation typically arises in computational solid mechanics, for modeling small deformations of compressible bodies. In 2D, piecewise cubic basis functions are used and the system of equations has

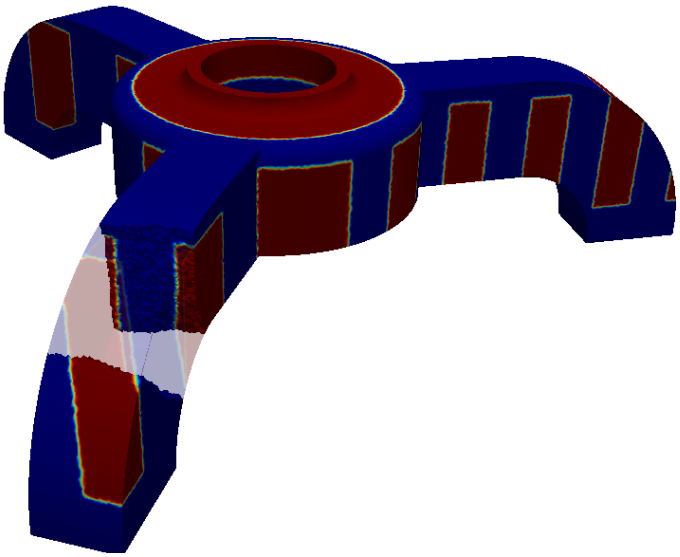
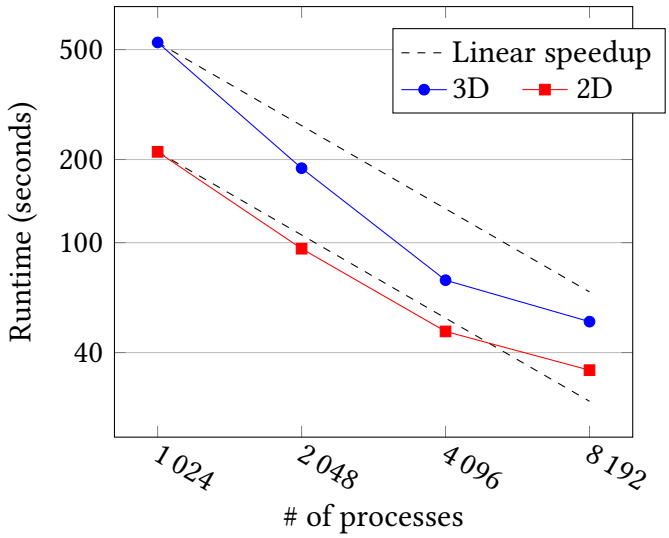


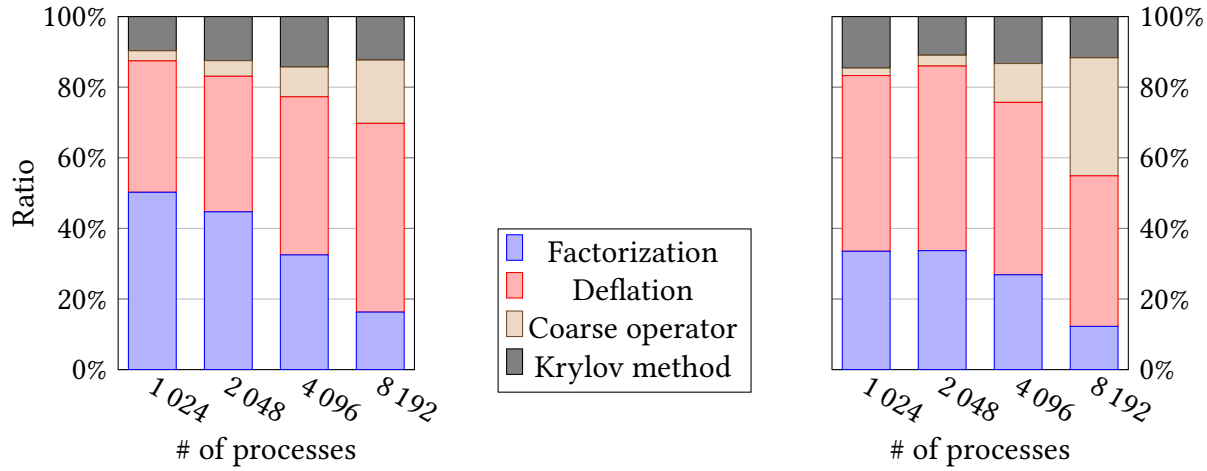
Fig 5.1: Variations of the material coefficients used for the three-dimensional strong scaling experiments.

approximately 33 nonzero entries per row. It is of constant size equal close to two billions unknowns. In 3D, piecewise quadratic basis functions are used and the system of equations has approximately 83 nonzero entries per row. The system is of constant size equal close to 300 million unknowns. These are so-called strong scaling experiments. Both geometries are displayed figs. A.1a and A.2a and partitioned with METIS. After the partitioning step, each local mesh is refined concurrently by splitting each triangle or tetrahedron into multiple smaller elements as done fig. 3.1. This means that the simulation starts with a relatively coarse global mesh (26 million triangles in 2D, 10 million tetrahedra in 3D), which is then refined in parallel (thrice in 2D, twice in 3D). A nice speedup is obtained from $1\,024 \times 2 = 2\,048$ to $8192 \times 2 = 16\,384$ threads as shown fig. 5.2a.



(a) Timings of various simulations

According to fig. 5.2c, the costly operations in the construction of the two-level preconditioner are the solution of each local eigenvalue problem eq. (1.40), column *Deflation*, and the factorization of each local solver $\{A_{ii}\}_{i=1}^{N}$, column *Factorization*. In 3D, the complexity of such operations typically grows superlinearly with respect to the number of unknowns, hence it is possible to achieve superlinear speedups. At peak performance, on 16 384 threads, the speedup relative to 2 048 threads equals $\frac{530.56}{51.76}$ which is approximately a tenfold decrease in runtime. In 2D, the local computation costs are lower, and tend to scale better with the number of unknowns, which makes it harder to achieve high speedups for larger numbers of subdomains. At peak performance, on 16 384 threads, the speedup relative to 2 048 threads equals $\frac{213.20}{34.54}$ which is approximately a sixfold decrease in runtime. In both cases, the solution of the eigenproblem is the limiting factor for achieving better speedups. This can be explained by the fact that the Lanczos [1950] method, on which ARPACK is based, tends to perform better for larger ratios $\left\{\frac{n_i}{\gamma_i}\right\}_{i=1}^{N}$, but these values decrease as subdomains get smaller. The local number of deflation vectors is uniform across subdomains and ranges from twenty to fifteen. For larger but fewer subdomains, the time to compute the solution, column *Solution*, i.e. the time for the GMRES to converge, is almost equal to the one spent in local forward eliminations and back substitutions. When the decompositions become bigger, subdomains are smaller, hence each local solution is computed faster and global communications have to be taken into account. To get a more precise idea of the communication-to-computation ratio, fig. 5.2b is quite useful, since the first two steps for computing the local factorizations and deflation vectors are purely concurrent and do not involve any communication. Thus, it is straightforward to get a lower bound of the aforementioned ratio. The time spent for assembling the coarse operator and for the Krylov method to converge is comprised of both communications and computations.



(b) Comparison of the time spent in various steps for building and using the preconditioner in 2D (left) and 3D (right).

	N	Factorization	Deflation	Solution	# of it.	Total	# of d.o.f.
3D	1 024	177.9 s	264.0 s	77.4 s	28	530.6 s	$293.98 \cdot 10^6$
	2 048	62.7 s	97.3 s	20.4 s	23	186.0 s	
	4 096	19.6 s	35.7 s	9.7 s	20	73.1 s	
	8 192	6.3 s	22.1 s	6.0 s	27	51.8 s	
2D	1 024	37.0 s	131.8 s	34.3 s	28	213.2 s	$2.14 \cdot 10^9$
	2 048	17.5 s	53.8 s	17.5 s	28	95.1 s	
	4 096	6.9 s	27.1 s	8.6 s	23	47.7 s	
	8 192	2.0 s	20.8 s	4.8 s	23	34.5 s	

(c) Breakdown of the timings used for the figure on top

Fig 5.2: Strong scaling experiments.

To assess the need for such a sophisticated preconditioner, the convergence histogram of a simple one-level method versus this two-level method is displayed fig. 5.3. One can easily understand that, while the cost of building the preconditioner cannot be neglected, it is necessary to ensure the convergence of the Krylov method: after more than 10 minutes, the one-level preconditioner barely decreases the relative error to 2×10^{-5}, while it takes 213.20 seconds for the two-level method to converge to the desired tolerance, cf. fig. 5.2c row #5. That is at least a threefold speedup. Because one-level methods are not numerically scalable as explained section 1.1.2, it is expected to get an even better speedup for larger decompositions.

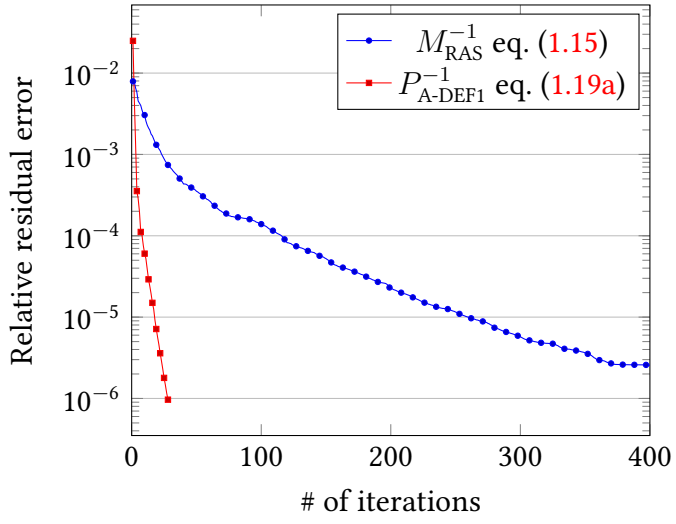


Fig 5.3:
Convergence of the restarted GMRES(40) for a 2D problem of linear elasticity using 1 024 subdomains. Timings for the setup and solution phases using $P_{\text{A-DEF1}}^{-1}$ are available in fig. 5.2, using M_{RAS}^{-1}, the convergence is not reached after 10 minutes.

Moving on to the weak scaling properties of the framework, the problem now being solved is a scalar equation of diffusivity with highly heterogeneous coefficients κ varying from 1 to $3 \cdot 10^6$ as displayed in fig. 5.4. The partitioned domain is $\Omega = [0;1]^d$ ($d = 2$ or 3) with piecewise quartic basis functions in 2D yielding linear systems with approximately 23 nonzero entries per row, and piecewise quadratic basis functions in 3D yielding linear systems with approximately 27 nonzero entries per row. This equation reads, find $u \in H_0^1(\Omega)$ such that:

$$\begin{aligned} -\nabla \cdot (\kappa \nabla u) &= 1 && \text{in } \Omega \\ u &= 0 && \text{on } [0;1] \times \{0\}\,. \end{aligned}$$

After using Green's formula, its variational formulation is, for all test functions $v \in H_0^1(\Omega)$:

$$\begin{aligned} a(u,v) &= \int_\Omega \kappa \nabla u \cdot \nabla v \\ l(v) &= \int_\Omega f \cdot v\,, \end{aligned}$$

where f is a source term. Such an equation typically arises for modeling flows in porous media, or in computational fluid dynamics. No change is needed in the framework, since all operations are algebraic. The only work needed outside of the framework is changing the mesh used for computations, as well as the variational formulation of the problem in FreeFem++ DSL. On average, there is a constant number of degrees of freedom per subdomain equal to roughly 280 thousands in 3D, and near 2.7 millions in 2D. As for the strong scaling experiment, after building and partitioning a global coarse mesh with few millions of elements, each local mesh is refined independently to ensure a constant size system per subdomain as the decomposition becomes bigger. The efficiency remains near the 90% mark, thanks to almost no variability in the time for the factorization of the local problems and for the construction of the deflation vectors. In 3D, the initial problem of 74 million unknowns is solved in 200 seconds on 512 threads. Using 16 384 threads, the problem is now made of approximately 2.3 billion unknowns, and it is solved in 215 seconds, which yields an efficiency of approximately 90%. In 2D, the initial problem of 695 million unknowns is solved in 175 seconds on 512 threads. Using 16 384 threads, the problem is now made of approximately 22.3 billions unknowns, and it is solved in 187 seconds, which yields an efficiency of approximately 96%. At such scales, the most penalizing step in the algorithm is the construction of the coarse operator, specially in 3D, with a nonnegligible increase in the time spent to assemble the Galerkin matrix E.

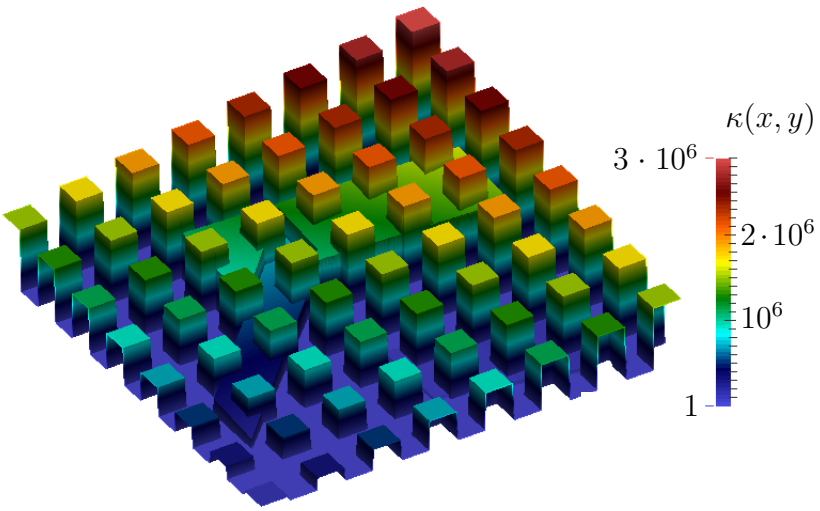
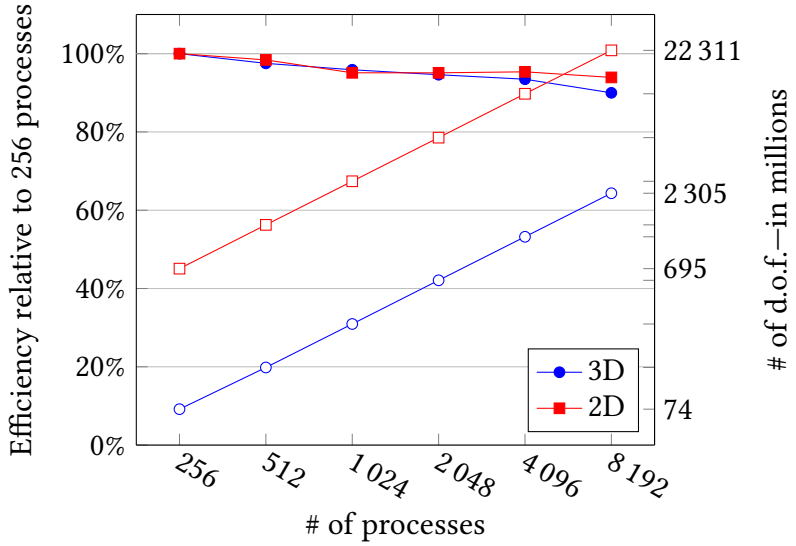
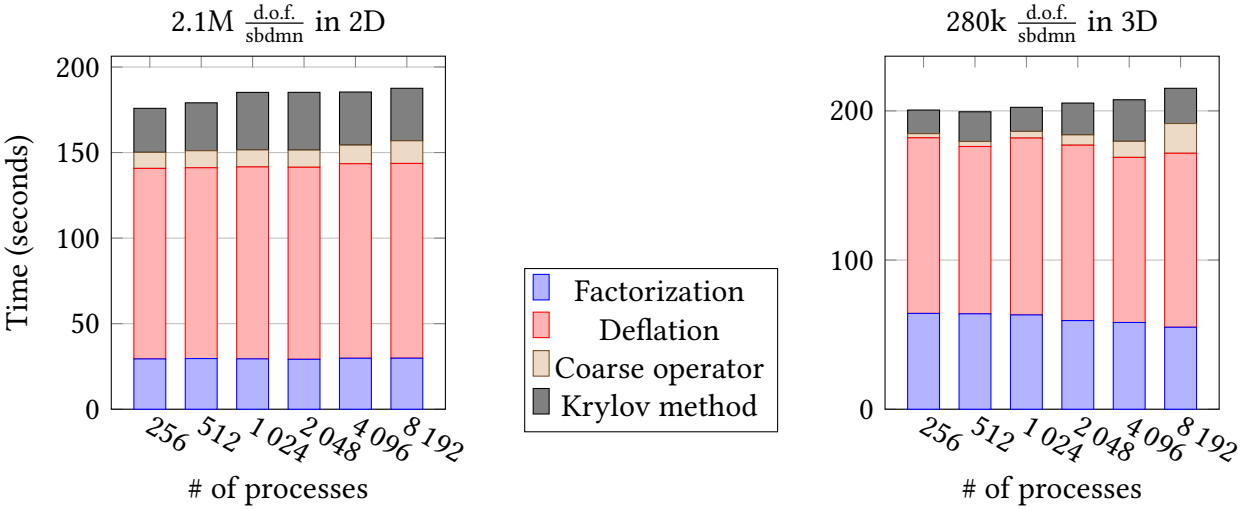


Fig 5.4: Diffusivity κ used for the two-dimensional weak scaling experiments with channels and inclusions.



(a) Timings of various simulations



(b) Comparison of the time spent in various steps for building and using the preconditioner

Eventually, the timings of the assembly and the factorization of the coarse operator E for all the previous simulations are now presented. Tables 5.2c and 5.5c already included these timings in their last column *Total* (> *Factorization* + *Deflation* + *Solution*), but for a more in-depth analysis, they are reported next separately. Figure 5.6 includes all timings relative to algorithms 4.1 and 4.2 described section 4.1: the construction of the communicators, the assembly of E, and its numerical factorization. The local number of deflation vectors computed and assembled in the coarse operator is once again uniform across subdomains. The most consuming part of the algorithm is the actual transfer and the assembly by the masters. Especially in 3D, when the coarse operator is becoming less and less sparse, a property that is directly linked with the average value of $\{\#\mathcal{O}_i\}_{i=1}^{N}$, it is likely to become a problem for even larger decomposition. This bottleneck is addressed in the context of algebraic multigrid methods [Falgout and Schroder 2014]. Note that the MPI implementation used for these experiments is not thread compliant and in particular, it does not support a level of thread support equal to `MPI_THREAD_MULTIPLE`, meaning that some unnecessary `#pragma omp critical` had to be used by the masters during the assembly. At these scales, another problem is the factorization of E. Indeed, increasing the number of masters P does not always have a beneficial effect for this concern because distributed solvers have

	N	Factorization	Deflation	Solution	# of it.	Total	# of d.o.f.
3D	256	64.2 s	117.7 s	15.8 s	13	200.6 s	$74.6 \cdot 10^6$
	512	64.0 s	112.2 s	19.9 s	18	199.4 s	$144.7 \cdot 10^6$
	1 024	63.2 s	118.6 s	16.2 s	14	202.4 s	$288.8 \cdot 10^6$
	2 048	59.4 s	117.6 s	21.3 s	17	205.3 s	$578.0 \cdot 10^6$
	4 096	58.1 s	110.7 s	27.9 s	20	207.5 s	$1.2 \cdot 10^9$
	8 192	55.0 s	116.6 s	23.6 s	17	215.2 s	$2.3 \cdot 10^9$
2D	256	29.4 s	111.3 s	25.7 s	29	175.8 s	$696.0 \cdot 10^6$
	512	29.6 s	111.5 s	28.0 s	28	179.1 s	$1.4 \cdot 10^9$
	1 024	29.4 s	112.2 s	33.6 s	28	185.2 s	$2.8 \cdot 10^9$
	2 048	29.2 s	112.2 s	33.7 s	28	185.2 s	$5.6 \cdot 10^9$
	4 096	29.8 s	113.7 s	31.0 s	26	185.4 s	$11.2 \cdot 10^9$
	8 192	29.8 s	113.8 s	30.7 s	25	187.6 s	$22.3 \cdot 10^9$

(c) Breakdown of the timings used for the figure on top

Fig 5.5: Weak scaling experiments.

difficulties scaling beyond 128 processes. The cost of factorization represents the maximum memory space needed by one master process for the linear solver to store the factorized matrix E^{-1}. This value is returned by MUMPS, which was the distributed solver used in these experiments.

N	P		dim(E)		$\#\mathcal{O}_i$ avg.		Cost of fact.		Time	
256	2		5 376		5.5		21 MB		9.39 s	
512	4		10 240		5.6		32 MB		9.96 s	
1 024	10	8	20 480	24 576	5.7	5.5	65 MB	57 MB	9.92 s	10.14 s
2 048	14	12	38 912	40 960	5.8	5.7	94 MB	83 MB	10.05 s	6.20 s
4 096	22	18	81 920	73 728	5.9	5.8	99 MB	73 MB	10.87 s	5.10 s
8 192	36	36	163 840	122 880	5.9	5.8	152 MB	118 MB	13.27 s	6.96 s

(a) Two-dimensional test cases

N	P		dim(E)		$\#\mathcal{O}_i$ avg.		Cost of fact.		Time	
256	4		5 120		11.5		38 MB		2.78 s	
512	6		10 240		12.4		78 MB		3.35 s	
1 024	8	8	20 480	22 528	13.0	12.0	156 MB	93 MB	4.42 s	11.25 s
2 048	12	12	40 960	40 960	13.8	12.9	332 MB	138 MB	6.91 s	5.68 s
4 096	18	22	73 728	73 728	14.2	13.7	434 MB	172 MB	10.75 s	8.04 s
8 192	64	48	131 072	131 072	14.7	14.6	420 MB	241 MB	19.92 s	17.30 s

(b) Three-dimensional test cases

Fig 5.6: Timings for assembling and factorizing the coarse operator in the two previous experiments. Results are gathered two-by-two, the first column with a gray overlay is for the diffusivity problem, the second column is for the elasticity problem.

5.2 Comparison with multigrid solvers

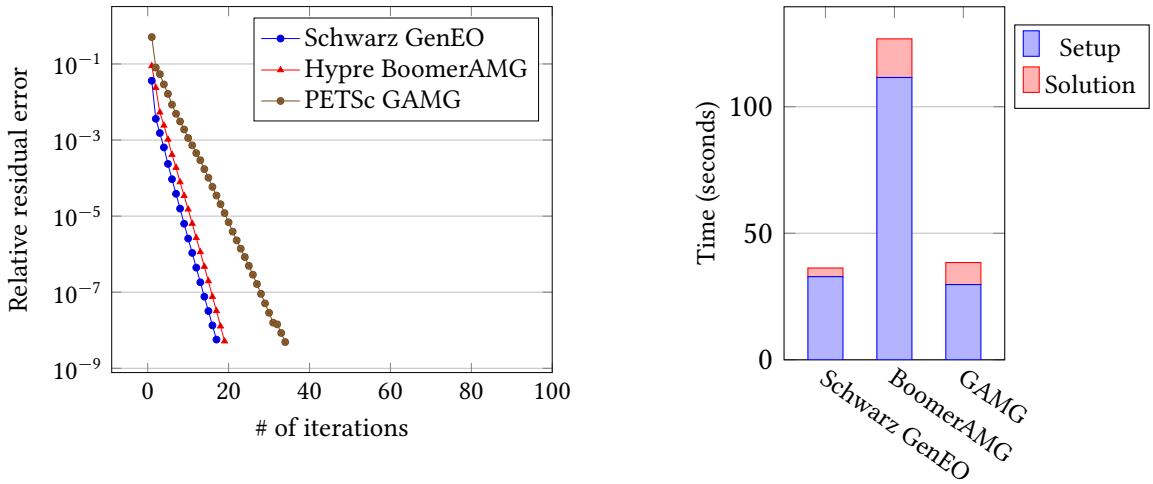
The framework will now be put to the test against both selective and aggregative algebraic multigrid solvers that are supposedly some of the most effective preconditioners on today's large-scale architectures: BoomerAMG [Baker et al. 2012] and GAMG, partially based on some previous work [Adams et al. 2004]. Once again, the domain decomposition preconditioner employed is a Restricted Additive Schwarz method with a minimal geometric overlap of one mesh element, enhanced with twenty deflation vectors per subdomain computed using eq. (1.40). The first problem solved is Poisson's three-dimensional eq. (1.5) discretized with piecewise linear finite elements on a square Ω using 4 096 processes and only one thread per process. The linear system has 217 million d.o.f. and is well conditioned since there is no heterogeneity. This results in a nice convergence of all three preconditioners as displayed fig. 5.7a. The parameters GAMG are:

- no relative threshold for dropping edges in the aggregation graphs,
- a one-dimensional near-null space set to a constant function.

The parameters for BoomerAMG are:

- Hybrid Modified Independent Set coarsening [De Sterck, U. M. Yang, and Heys 2006],
- extended classical interpolation [De Sterck, Falgout, et al. 2008],
- no C-F relaxation,
- two levels of aggressive coarsening,
- truncated interpolation to four entries per row.

While these are recommended for best practices and were also suggested by the hypre team via email, they resulted in slower convergence as shown fig. 5.7b. When it comes down to the overlapping Schwarz preconditioner and the smoothed aggregation multigrid preconditioner, they perform similarly. The setup cost of the domain decomposition method is slightly higher, but there is less communication involved when applying this preconditioner during the iterative method because the grid hierarchy is much simpler, therefore the solution step is a bit faster than with GAMG.



(a) Convergence of the preconditioner GMRES

(b) Timings broken down into two steps

Fig 5.7: Comparison of iterative solvers for solving Poisson's equation.

The second problem is the system of linear elasticity, cf. eq. (5.1), with highly heterogeneous coefficients and discretized using piecewise quadratic finite elements, yielding a linear system of 262 million d.o.f. The parameters for the domain decomposition preconditioners are exactly the same as before, whereas those for GAMG are now:

- a relative threshold for dropping edges in the aggregation graphs of 0.2,
- a six-dimensional near-null space set to the rigid body modes—three translations and three rotations.

Another choice for the threshold—0.25—led to a slower convergence, but a more robust preconditioner, cf. fig. 5.8. Adequately tuning GAMG was done thanks to fruitful conversations and exchanges with Jed Brown (Argonne National Laboratory), Luke Olson (University of Illinois at Urbana-Champaign), and Mark Adams (Lawrence Berkeley National Laboratory). One key issue that had to be untangled is the fact that matrices had some extremely large values on their diagonal when penalizing Dirichlet boundary conditions, as done in FreeFem++. This lead to a bad coarsening with the smoothed aggregation of GAMG. Obviously this is not a problem for domain decomposition preconditioners with direct solvers used inside each subdomain since they can handle such a penalization[3]. Switching to another approach for imposing boundary conditions, among other adjustments that they suggested, led to better results displayed in this thesis. The remark made in the previous comparison still holds: the setup (resp. convergence to the solution) is slightly faster (resp. slower) with GAMG than with the overlapping Schwarz method.

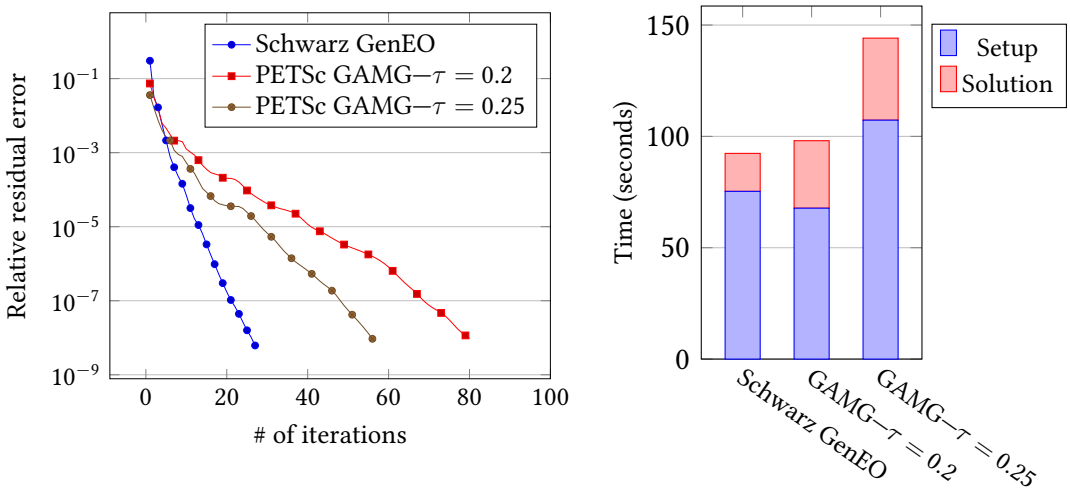


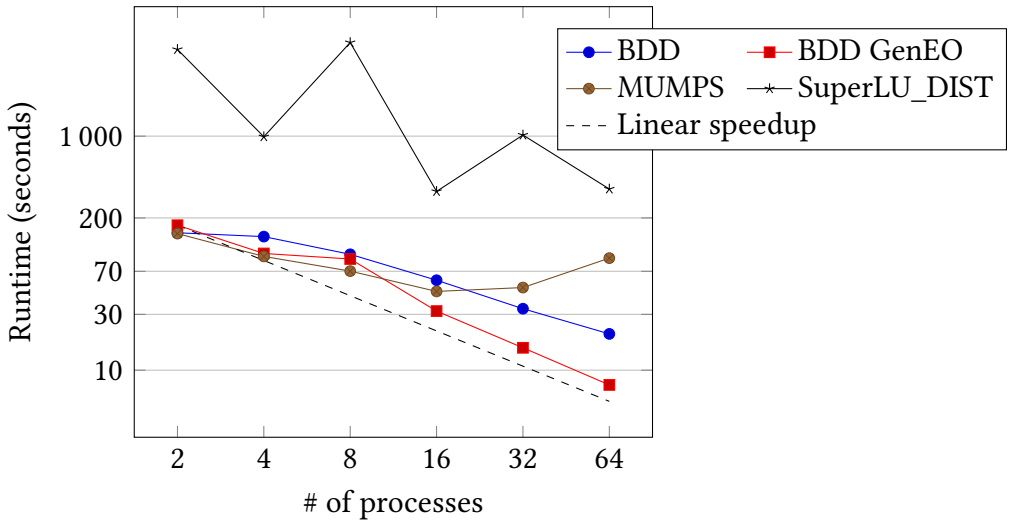
Fig 5.8: Comparison of iterative solvers for solving the system of linear elasticity.

5.3 Comparison with direct solvers

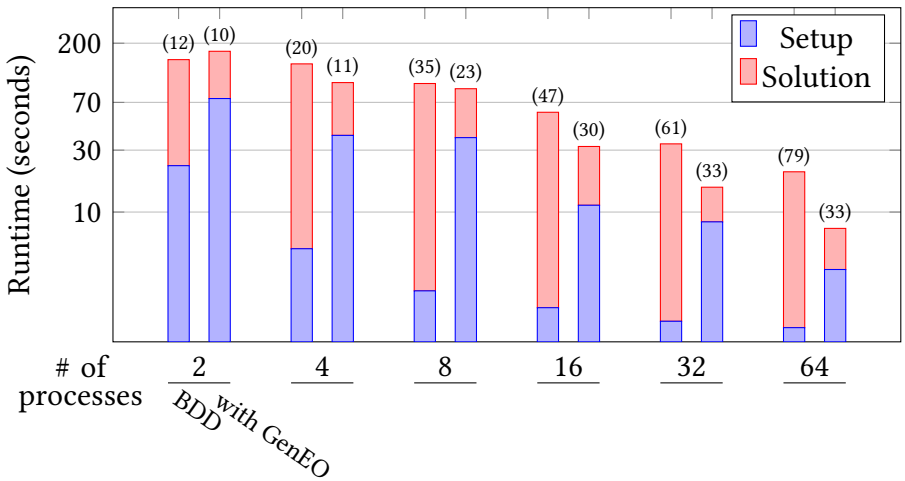
Eventually, the framework is now compared on much smaller decompositions with direct solvers: MUMPS and SuperLU_DIST [Li and Demmel 2003]. This time, the domain decomposition preconditioners are the BDD method, as well as the BDD method enhanced with deflation vectors computed using eq. (1.43). The finite element backend was FreeFem++, but the same computations could be carried out with Feel++. The problem being solved

[3]Out of all the linear solvers presented section 2.1.3, PaStiX was the only one having slight issues and performing unnecessary pivoting during the numerical factorization of the local problems with Dirichlet boundary conditions. Thanks to the help of Xavier Lacoste and Pierre Ramet, tuning the parameter `DPARM_EPSILON_MAGN_CTRL` fixed this.

is Poisson's two-dimensional eq. (1.5) discretized with piecewise quartic finite elements on a square Ω. The linear system has five million d.o.f. before substructuration and the simulation is run on a much less efficient hardware architecture than the one of Curie, a small NUMA system hosted at Laboratoire Jacques-Louis Lions and assembled by SGI. Direct solvers spend much of their time in the setup phase. For substructuring methods, with the standard BDD method, the local Schur complements needed by the global Schur complement eq. (1.28) are not computed explicitly. In this experiment, MUMPS was used to factorize in each subdomain $j \in [\![1; N]\!]$ local Dirichlet matrices $\left\{ \mathring{A}_{ii}^{(j)^{-1}} \right\}_{j=1}^{N}$. It turned out that the consecutive matrix multiplications and forward eliminations and back substitutions, cf. definition 1.11, were computed rather slowly, as displayed in fig. 5.9b. On the other hand, when using the BDD method enhanced with GenEO deflation vectors, the local Schur complements must be retrieved explicitly for the local generalized eigenvalue problems 1.43. In that case, the setup phase is likely to be slower, but the solution phase is much quicker, since the local workload for applying the Schur complement decreases from three sparse matrix-vector product and one forward elimination and back substitution to only a BLAS call to `?symv`, cf. section 2.1.2.



(a) Comparison of runtimes



(b) Breakdown of the timings of the substructuring preconditioners for the figure on top. The number in smaller font above each bar is the number of CG iterations

Fig 5.9: Comparison with direct solvers for solving Poisson's equation.

Substructuring methods tend to be much less robust than overlapping Schwarz methods when using automatic graph partitioners, because the former sometimes behaves erratically in presence of rough interfaces between subdomains, see [Klawonn, Rheinbach, and Widlund 2008] for an in-depth justification. For that reason, they are no additional numerical results with substructuring methods. A strategy introduced by Samake [2014] for alleviating this problem consists in:

- creating a really coarse mesh with as many elements as subdomains,
- then refining each superelement concurrently.

The first step ensures that interfaces between subdomains are smooth enough, in fact they are either connected through the vertex or a linear edge or planar face of a superelement, while the second step makes it possible to perform large-scale experiments. Because this method was first used in conjunction with the mortar method [Bernardi, Maday, and Patera 1993], there is no concern about conformity of each local mesh through the interfaces, this is however a necessity for the substructuring preconditioners introduced in section 1.2.

Conclusion

While recent hardware advances in the parallel computing community have led to a greater and greater level of concurrency, it is not always easy for scientists using implicit solvers in their software to efficiently handle such architectures. In this context, applied mathematicians have looked into the "divide & conquer" paradigm to accelerate numerical simulations. The past decades have brought numerous efficient preconditioners based on domain decomposition methods. In this thesis, two of the most prominent methods, the overlapping Schwarz preconditioners and the substructuring preconditioners have been unified into an abstract parallel framework. Manipulating algebraic equalities, it was established how it is possible to perform extreme-scale experiments with domain decomposition methods, even with a sequential discretization kernel like FreeFem++, a software for finite element discretizations. At those scales, it is of paramount importance to ensure that the preconditioners are numerically robust. For domain decomposition methods, this is usually done by introducing an auxiliary problem known as a coarse problem. Its goal is to couple all subdomains efficiently in order to guarantee a fast convergence of preconditioned iterative methods, in terms of number of iterations. This coupling introduces strong data dependencies and tends to diminish the algorithmic performance of the said preconditioners. The novel construction of the coarse operator presented in this thesis shows great scalability on Curie, a Tier-0 supercomputer for PRACE. A comparison with multigrid and direct solvers is also carried out. It shows that the framework presented has comparable or better performances than state of the art third-party libraries. Most of the results presented in this document have already led to publications, but they also raised some important new points that are described in the following paragraph.

Perspectives and open questions

- When looking at overlapping Schwarz methods, the two methods studied were the additive Schwarz method and the restricted additive Schwarz method. It is sometimes useful to switch to optimized Schwarz methods which have more sophisticated transmission conditions and can solve harder problems, as in [Nourtier-Mazauric and Blayo 2010]. Moreover, as described by St-Cyr, Gander, and Thomas [2007], they follow, from an algebraic point of view, the same workflow as standard overlapping Schwarz methods.

- The construction of the coarse operator has a level of abstraction high enough so that it is easy for an end-user to try any kind of deflation vector, as well as change the definition of the Galerkin operator. The framework may thus validate other preconditioners based on deflation techniques, such as a recent one introduced for the Helmholtz equation [Conen et al. 2014].

- While the GenEO approach based on the solution of local generalized eigenvalue problems leads to robust preconditioners, the time spent in solving these eigenvalue

problems is not neglectable. There are then at least two possible directions of research to circumvent this bottleneck.

  - Mixed precision could lead to less computations for the eigensolvers and for the assembly of the coarse operator, as it is the case in most scientific computations, e.g. [Baboulin et al. 2009]. Indeed, the framework currently only deals with double-precision floating-point scalars, but it might be interesting to see the numerical and algorithmic impact of switching to single-precision floating-point deflation vectors for instance.
  - Adaptive methods without the a priori construction of the eigenvectors are also appealing. This is already applied in the multigrid community, cf. [Brandt et al. 2011; Brezina et al. 2004], where some preconditioners are able to adjust the coarsening process throughout the convergence of an iterative method.

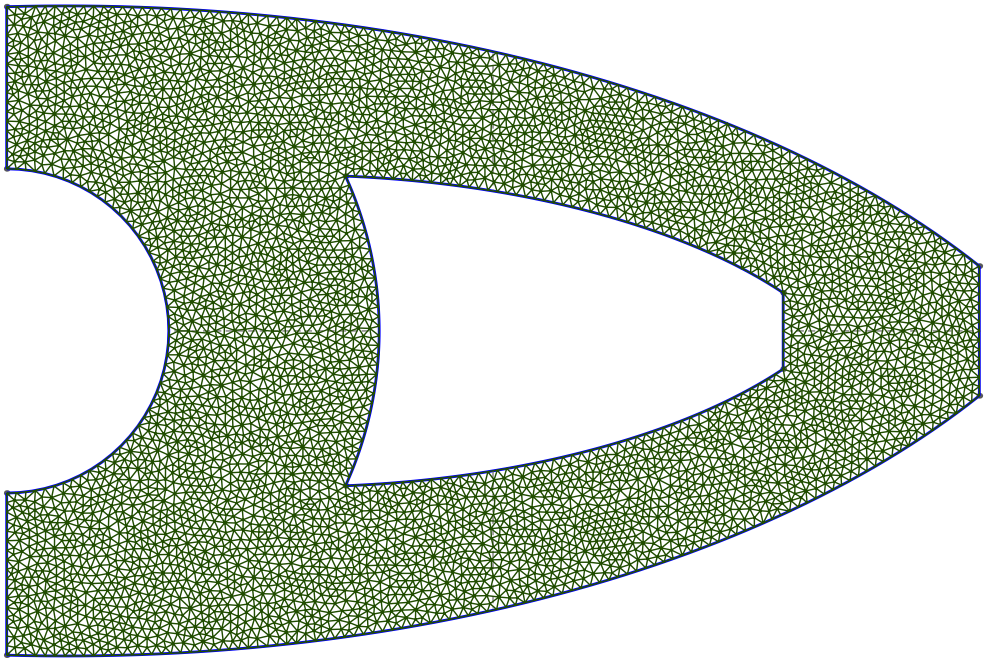
- In the context of nonlinear iterations or unsteady problems, it is not clear if it is possible to recycle the coarse space, see [Gosselet, Rey, and Pebrel 2013; Parks et al. 2006] for examples. On the one hand, it would be beneficial to avoid the computation of new eigenvectors at each iteration, but on the other hand, could it still be possible to have condition number estimates for successive linear systems ?

- A follow up question concerns adaptive mesh refinement (AMR), cf. [Berger and Colella 1989; Deiterding 2005], which brings another form of variability when solving multiple linear systems. The approach presented in this thesis is particularly helpful for assembling large problems when only a sequential mesher or grid generator is available. However, if AMR is used throughout a simulation, it is currently not possible to ensure a proper load-balancing between subdomains. Even though this limitation is more directly tied to the discretization library that handles the mesh or grid structures, this is an important point to keep in mind.

- Eventually, for achieving sustainable performance at even larger scale, if targeting for example simulations with up to one hundred thousand subdomains, a multilevel extension of the current coarse operator is necessary. However, such extensions in the domain decomposition community tend to lead to poor algorithmic performances, cf. [Šístek et al. 2013]. A modification of the structure of the Galerkin operator might also lead to less inter-process interactions [Falgout and Schroder 2014], which in the end would increase the scalability of the various steps related to deflation in the construction of the preconditioner.

Appendices

A Gmsh geometries

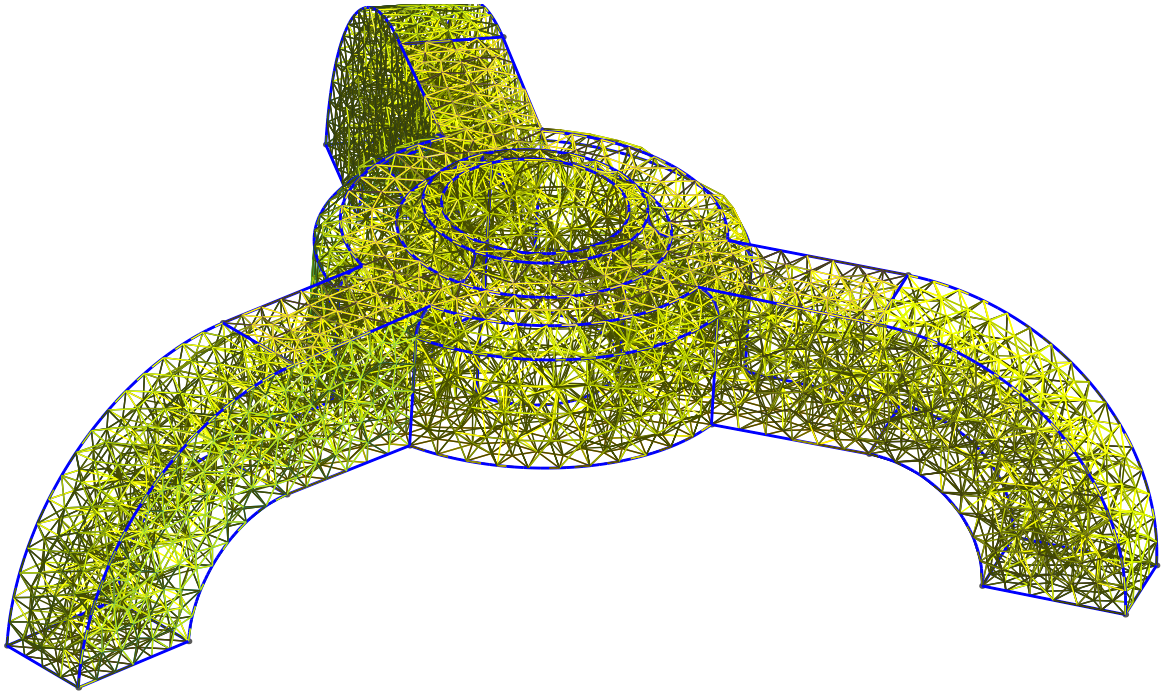


(a) Generated mesh

```
height = 1;
length = 1.5;
offset = 0.5;
tip = 0.2;
lc = 0.015;
ratio = 0.8;
Point(1) = {0, 0, 0, lc};
Point(2) = {0, height, 0, lc};
Point(3) = {0, height-offset/2, 0, lc};
Point(4) = {0, offset/2, 0, lc};
Point(5) = {length, (height+tip)/2, 0, lc};
Point(6) = {length, (height-tip)/2, 0, lc};
Point(7) = {0, height/2, 0, lc};
Circle(1) = {4, 7, 3};
Point(8) = {2.5, 0.1, -0, lc};
Point(9) = {2.5, 0.9, -0, lc};
Ellipse(3) = {2, 1, 8, 5};
Ellipse(4) = {1, 2, 9, 6};
Line(5) = {3, 2};
Line(6) = {4, 1};
Line(7) = {6, 5};
Dilate {{0.95, 0.525, 0}, 0.45} {
  Duplicata { Line{3}; }
}
Dilate {{0.95, 0.475, 0}, 0.45} {
  Duplicata { Line{4}; }
}
Translate {0, -0.1, 0} {
  Point{15};
}
Translate {0, 0.1, 0} {
  Point{11};
}
Line(10) = {13, 17};
Ellipse(11) = {14, 7, 11, 10};
Line Loop(12) = {3, -7, -4, -6, 1, 5};
Line Loop(13) = {8, 10, -9, 11};
Plane Surface(14) = {12, 13};
Physical Line(15) = {5, 6};
Physical Line(16) = {7};
```

(b) Input `.geo` file

Fig A.1: A two-dimensional cantilever.



(a) Generated mesh

```
lc = 0.025;
offset = 0.075;
ratio = 1.;
Point(1) = {offset, 0.0, 0, lc};
Point(2) = {offset+0.1, 0.0, 0, lc};
Point(3) = {offset+0.1, 0.1, 0, lc};
Line(2) = {1, 2};
Line(3) = {2, 3};
Point(5) = {offset+0.08, 0.1, 0, lc*ratio};
Point(6) = {offset+0.08, 0.12, 0, lc*ratio};
Circle(4) = {3, 5, 6};
Point(7) = {offset+0.015, 0.14, 0, lc*ratio};
Point(8) = {offset+0.035, 0.12, 0, lc*ratio};
Point(9) = {offset+0.035, 0.14, 0, lc*ratio};
Point(10) = {offset, 0.14, 0, lc*ratio};
Line(5) = {6, 8};
Line(6) = {7, 10};
Line(7) = {10, 1};
Circle(8) = {8, 9, 7};
Line Loop(9) = {8, 6, 7, 2, 3, 4, 5};
Plane Surface(10) = {9};
angle = Pi/12;
Rotate {{0, 1, 0}, {0, 0, 0}, angle} {
    Surface{10};
}
Extrude {{0, 1, 0}, {0, 0, 0}, 2*Pi/3-2*angle} {
    Surface{10};
}
Rotate {{0, 1, 0}, {0, 0, 0}, 2*angle} {
    Duplicata { Surface{47}; }
}
Extrude {{0, 1, 0}, {0, 0, 0}, 2*Pi/3-2*angle} {
    Surface{48};
}
Rotate {{0, 1, 0}, {0, 0, 0}, 2*angle} {
    Duplicata { Surface{92}; }
}
Extrude {{0, 1, 0}, {0, 0, 0}, 2*Pi/3-2*angle} {
    Surface{93};
}
length = 0.3;
```

```
Point(173) = {length, 0, (offset+0.035)*Sin[3/2*angle], lc*length*3};
Point(174) = {length, 0, -(offset+0.035)*Sin[3/2*angle], lc*length*3};
Line(278) = {168, 173};
Line(279) = {173, 174};
Line(280) = {174, 2};
Translate {0, 0.12, 0} {
    Duplicata { Point{174}; }
}
Translate {0, 0.12, 0} {
    Duplicata { Point{173}; }
}
Line(281) = {6, 178};
Line(282) = {178, 174};
Line(283) = {179, 173};
Line(284) = {179, 178};
Line(287) = {179, 177};
Line Loop(288) = {283, 279, -282, -284};
Plane Surface(289) = {288};
height = 0.1;
Extrude {{0, 0, 1}, {length, -height, 0}, -Pi/2} {
  Surface{289};
}
Delete {
  Volume{4};
}
Delete {
  Surface{289, 298, 306};
}
Delete {
  Line{283, 282};
}
Delete {
  Volume{3, 1, 2};
}
Delete {
  Surface{137, 10, 93, 92, 47, 48};
}
Delete {
  Surface{124, 34, 79, 46, 136, 91, 26, 71, 116};
```

```
}
Delete {
  Line{97, 60, 105, 2, 108, 5, 100, 63, 55, 18, 15, 52,
       50, 13, 103, 6, 95, 58};
}
Delete {
  Surface{112, 75, 67, 22, 30, 120};
}
Delete {
  Line{57, 94, 96, 59, 49, 51, 14, 12, 8, 7, 102, 104};
}
Circle(312) = {112, 49, 136};
Circle(313) = {108, 49, 132};
Circle(314) = {106, 47, 130};
Circle(315) = {84, 49, 13};
Circle(316) = {88, 49, 17};
Circle(317) = {82, 47, 11};
Circle(318) = {160, 49, 10};
Circle(319) = {156, 49, 7};
Circle(320) = {154, 47, 8};
Delete {
  Line{110, 320, 20, 21, 25, 319, 111, 115, 318, 312, 313,
       314, 70, 78, 66, 65, 317, 315, 316};
}
Delete {
  Point{154, 106, 11, 13, 17, 156, 160, 112, 108};
}
Circle(316) = {10, 49, 136};
Circle(317) = {136, 49, 88};
Circle(318) = {88, 49, 10};
Circle(319) = {132, 49, 84};
Circle(320) = {84, 49, 7};
Circle(321) = {7, 49, 132};
Circle(322) = {130, 47, 82};
Circle(323) = {82, 47, 8};
Circle(324) = {8, 47, 130};
Line Loop(325) = {323, 324, 322};
Line Loop(326) = {320, 321, 319};
Circle(327) = {84, 83, 82};
Circle(328) = {132, 131, 130};
Circle(329) = {7, 9, 8};
Line Loop(330) = {323, -329, -320, 327};
Ruled Surface(331) = {330};
Line Loop(332) = {322, -327, -319, 328};
Ruled Surface(333) = {332};
Line Loop(334) = {328, -324, -329, 321};
Ruled Surface(335) = {334};
Line Loop(336) = {318, 316, 317};
Plane Surface(337) = {326, 336};
Rotate {{0, 1, 0}, {0, 0, 0}, 2*Pi/3} {
  Duplicata { Line{287, 281, 278, 280, 279, 301, 297, 296,
       305, 284, 293, 292, 291, 294}; }
}
Rotate {{0, 1, 0}, {0, 0, 0}, -2*Pi/3} {
  Duplicata { Line{287, 281, 278, 280, 279, 301, 297, 296,
       305, 284, 293, 292, 291, 294}; }
}
Line Loop(366) = {355, 98, 99, 353, 360, -362, -357};
Plane Surface(367) = {366};
Line Loop(368) = {354, 358, -364, -359, 352, -62, -61};
Plane Surface(369) = {368};
Line Loop(370) = {341, 53, 54, 339, 346, -348, -343};
Plane Surface(371) = {370};
Line Loop(372) = {345, 350, -344, -340, 16, 17, -338};
Plane Surface(373) = {372};
Line Loop(374) = {280, 3, 4, 281, 305, -293, -301};
Plane Surface(375) = {374};
Line Loop(376) = {296, 291, -297, -278, 106, 107, -287};
Plane Surface(377) = {376};
Line Loop(378) = {364, 363, 362, 365};
Plane Surface(379) = {378};
Line Loop(380) = {348, 351, 350, 349};
Plane Surface(381) = {380};
Line Loop(382) = {357, -363, -358, 356};
Ruled Surface(383) = {382};
Line Loop(384) = {360, 365, -359, 361};
Ruled Surface(385) = {384};
Line Loop(386) = {346, 351, -345, 347};
Ruled Surface(387) = {386};
Line Loop(388) = {343, -349, -344, 342};
Ruled Surface(389) = {388};
Delete {
  Line{74, 29, 119};
}
Delete {
  Point{116, 21, 164, 12, 155, 107};
}
Circle(390) = {140, 65, 92};
Circle(391) = {92, 65, 1};
Circle(392) = {1, 65, 140};
Line(393) = {136, 140};
Line(394) = {88, 92};
Line(395) = {10, 1};
Line Loop(396) = {353, -361, 352, -86, 339, -347, 338,
      -41, 281, -284, 287, -131};
Plane Surface(397) = {325, 396};
Line Loop(398) = {78, 354, 356, 355, 123, 278, 279, 280,
      33, 340, 342, 341};
Line Loop(399) = {390, 391, 392};
Plane Surface(400) = {398, 399};
Line Loop(401) = {395, -391, -394, 318};
Ruled Surface(402) = {401};
Line Loop(403) = {394, -390, -393, 317};
Ruled Surface(404) = {403};
Line Loop(405) = {392, -393, -316, 395};
Ruled Surface(406) = {405};
Surface Loop(407) = {83, 371, 400, 369, 383, 367, 128,
      132, 397, 331, 335, 333, 337, 404, 402, 406, 377,
      310, 311, 302, 375, 38, 42, 373, 387, 381, 389,
      87, 385, 379};
Volume(408) = {407};
Physical Surface(409) = {311, 381, 379};
Physical Surface(410) = {367, 383, 369, 385, 331, 333,
      335, 397, 38, 42, 377, 310, 375, 302, 400, 132,
      128, 83, 87, 371, 373, 389, 387};
Physical Surface(411) = {406, 404, 402, 337};
Physical Volume(412) = {408};
```

(b) Input `.geo` file

Fig A.2: A three-dimensional tripod.

B Structural UML diagram of the library

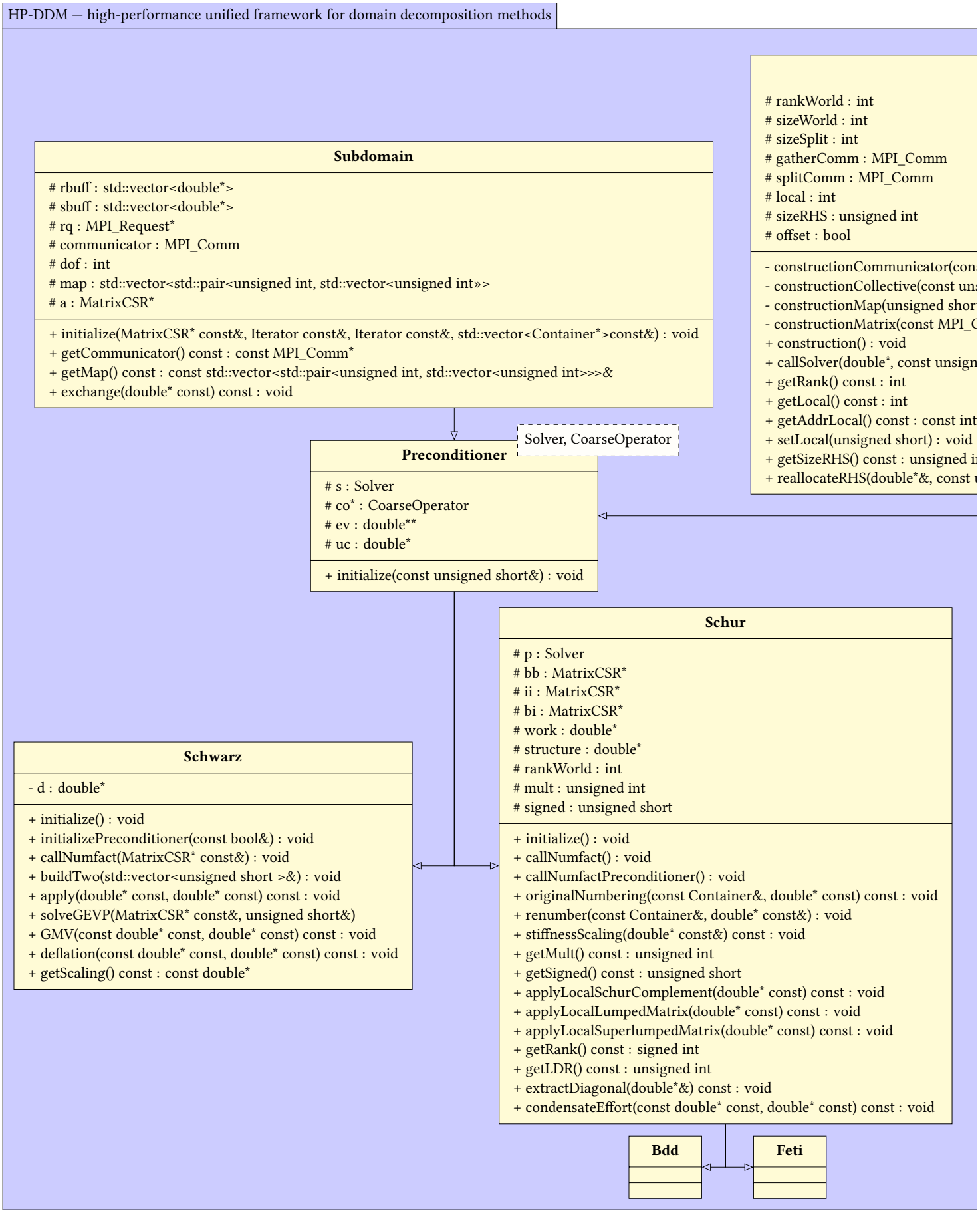
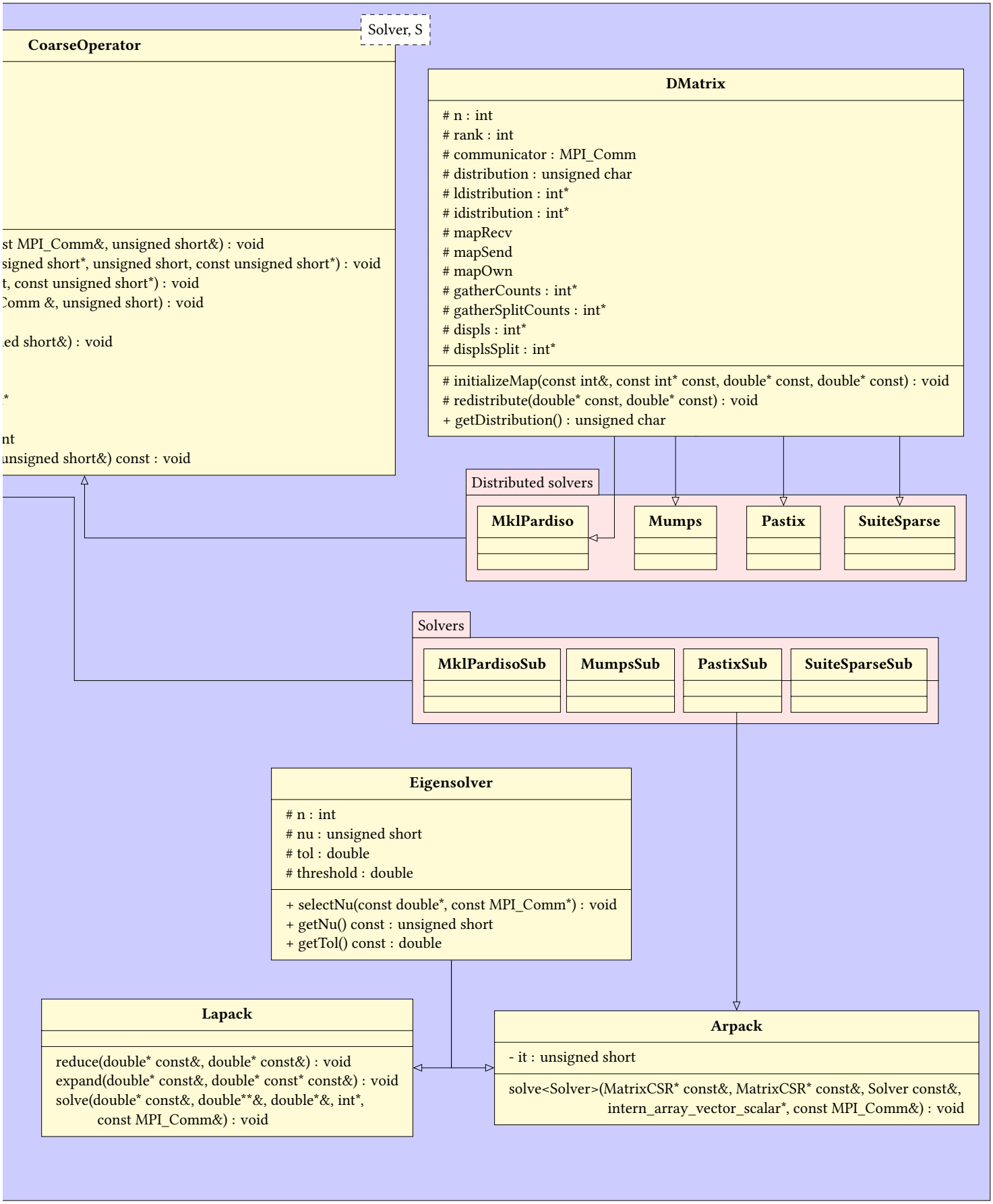


Fig B.1: Structural UML diagram of the library.


Solver, S
CoarseOperator
st MPI_Comm&, unsigned short&) : void
signed short*, unsigned short, const unsigned short*) : void
t, const unsigned short*) : void
Comm &, unsigned short) : void
ed short&) : void
*
nt
unsigned short&) const : void
DMatrix
n : int
rank : int
communicator : MPI_Comm
distribution : unsigned char
ldistribution : int*
idistribution : int*
mapRecv
mapSend
mapOwn
gatherCounts : int*
gatherSplitCounts : int*
displs : int*
displsSplit : int*
initializeMap(const int&, const int* const, double* const, double* const) : void
redistribute(double* const, double* const) : void
+ getDistribution() : unsigned char
Distributed solvers
MklPardiso
Mumps
Pastix
SuiteSparse
Solvers
MklPardisoSub
MumpsSub
PastixSub
SuiteSparseSub
Eigensolver
n : int
nu : unsigned short
tol : double
threshold : double
+ selectNu(const double*, const MPI_Comm*) : void
+ getNu() const : unsigned short
+ getTol() const : double
Lapack
reduce(double* const&, double* const&) : void
expand(double* const&, double* const* const&) : void
solve(double* const&, double**&, double*&, int*,
const MPI_Comm&) : void
Arpack
- it : unsigned short
solve<Solver>(MatrixCSR* const&, MatrixCSR* const&, Solver const&,
intern_array_vector_scalar*, const MPI_Comm&) : void

C Two published papers

Here are included two papers published during this thesis.

- The first one published in Journal of Numerical Mathematics [Jolivet, Dolean, et al. 2012] gathers some results presented during the third workshop on FreeFem++.
- The second one presented during the Supercomputing conference in 2013 [Jolivet et al. 2013] was one of the six papers nominated for the best paper award out of 90 accepted papers out of 457 submissions. It was republished in [Jolivet et al. 2014b].

Sorry, this version does not include the aforementioned papers.
You can access them at the following URLs:

- http://www.degruyter.com/view/j/jnma.2012.20.issue-3-4/jnum-2012-0015/jnum-2012-0015.xml,
- http://dl.acm.org/citation.cfm?id=2503212.

Bibliography

Adams, M., H. Bayraktar, T. Keaveny, and P. Papadopoulos (2004). "Ultrascalable Implicit Finite Element Analyses in Solid Mechanics with over a Half a Billion Degrees of Freedom". In: *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. SC04. IEEE Computer Society, 34:1–34:15 (cit. on p. 98).

Alimi, J-M., V. Bouillot, Y. Rasera, V. Reverdy, P-S. Corasaniti, I. Balmes, S. Requena, X. Delaruelle, and J-N. Richet (2012). "First-ever full observable universe simulation". In: *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*. SC12. IEEE Computer Society (cit. on pp. 1, 6).

Amestoy, P., I. Duff, J-Y. L'Excellent, and J. Koster (2001). "A fully asynchronous multifrontal solver using distributed dynamic scheduling". In: *SIAM Journal on Matrix Analysis and Applications* 23.1, pp. 15–41. URL: http://mumps.enseeiht.fr/ (cit. on p. 40).

Amestoy, P., A. Guermouche, J-Y. L'Excellent, and S. Pralet (2006). "Hybrid scheduling for the parallel solution of linear systems". In: *Parallel Computing* 32.2, pp. 136–156 (cit. on p. 40).

Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammerling, A. McKenney, and D. Sorensen (1999). *LAPACK Users' Guide*. Vol. 9. SIAM (cit. on p. 41).

Arnoldi, W. (1951). "The principle of minimized iterations in the solution of the matrix eigenvalue problem". In: *Quarterly of Applied Mathematics* 9.1, pp. 17–29 (cit. on p. 41).

Baboulin, M., A. Buttari, J. Dongarra, J. Kurzak, Julie Langou, Julien Langou, P. Luszczek, and S. Tomov (2009). "Accelerating scientific computations with mixed precision algorithms". In: *Computer Physics Communications* 180.12, pp. 2526–2533 (cit. on p. 104).

Babuška, I. and M. Suri (1992). "Locking effects in the finite element approximation of elasticity problems". In: *Numerische Mathematik* 62.1, pp. 439–463 (cit. on p. 92).

Badia, S., A. Martín, and J. Principe (2013). "Enhanced balancing Neumann-Neumann preconditioning in computational fluid and solid mechanics". In: *International Journal for Numerical Methods in Engineering* 96.4, pp. 203–230 (cit. on p. 86).

Badia, S., A. Martín, and J. Príncipe (2013). "Implementation and scalability analysis of balancing domain decomposition methods". In: *Archives of Computational Methods in Engineering* 20.3, pp. 239–262 (cit. on p. 82).

Baker, A., R. Falgout, T. Kolev, and U. M. Yang (2012). "Scaling *hypre*'s multigrid solvers to 100,000 cores". In: *High-Performance Scientific Computing*. Springer, pp. 261–279 (cit. on p. 98).

Balaji, P., W. Bland, W. Gropp, R. Latham, H. Lu, A. Peña, K. Raffenetii, R. Thakur, and J. Zhang (2014). *MPICH User's Guide*. Tech. rep. Argonne National Laboratory. URL: http://www.mpich.org/documentation/guides/ (cit. on p. 86).

Balay, S., M. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. Curfman McInnes, K. Rupp, B. Smith, and H. Zhang (2014). *PETSc Users*

Manual. Tech. rep. ANL-95/11 - Revision 3.5. Argonne National Laboratory. URL: http://www.mcs.anl.gov/petsc (cit. on p. 42).

Balay, S., W. Gropp, L. Curfman McInnes, and B. Smith (1997). "Efficient management of parallelism in object-oriented numerical software libraries". In: *Modern Software Tools in Scientific Computing*, pp. 163–202 (cit. on pp. 1, 6, 66).

Bangerth, W., R. Hartmann, and G. Kanschat (2007). "deal.II—a general-purpose object-oriented finite element library". In: *ACM Transactions on Mathematical Software* 33.4, pp. 24–27 (cit. on p. 42).

Bastian, P., M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander (2008). "A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE". In: *Computing* 82.2–3, pp. 121–138 (cit. on p. 43).

Bastian, P., M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander (2008). "A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework". In: *Computing* 82.2–3, pp. 103–119 (cit. on p. 43).

Berger, M. and P. Colella (1989). "Local adaptive mesh refinement for shock hydrodynamics". In: *Journal of Computational Physics* 82.1, pp. 64–84 (cit. on p. 104).

Bernardi, C., Y. Maday, and A. Patera (1993). "Domain decomposition by the mortar element method". In: *Asymptotic and Numerical Methods for Partial Differential Equations with Critical Parameters*. Vol. 384. NATO Series C. Springer, pp. 269–286 (cit. on p. 101).

Bhardwaj, M., D. Day, C. Farhat, M. Lesoinne, K. Pierson, and D. Rixen (2000). "Application of the FETI method to ASCI problems—scalability results on 1000 processors and discussion of highly heterogeneous problems". In: *International Journal for Numerical Methods in Engineering* 47.1-3, pp. 513–535 (cit. on p. 81).

Bhardwaj, M., K. Pierson, G. Reese, T. Walsh, D. Day, K. Alvin, J. Peery, C. Farhat, and M. Lesoinne (2002). "Salinas: A scalable software for high-performance structural and solid mechanics simulations". In: *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. SC02. IEEE Computer Society, pp. 1–19 (cit. on p. 81).

Bhatele, A., P. Jetley, H. Gahvari, L. Wesolowski, W. Gropp, and L. Kale (2011). "Architectural constraints to attain 1 Exaflop/s for three scientific application classes". In: *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, pp. 80–91 (cit. on p. 87).

Biros, G. and O. Ghattas (2005). "Parallel Lagrange–Newton–Krylov–Schur Methods for PDE-Constrained Optimization. Part I: The Krylov–Schur Solver". In: *SIAM Journal on Scientific Computing* 27.2, pp. 687–713 (cit. on pp. 3, 8).

Blackford, S., A. Petitet, R. Pozo, K. Remington, R. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. (2002). "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2, pp. 135–151 (cit. on p. 38).

Boiteau, O. (2009). *Parallélisme et décomposition de domaine : la méthode FETI*. R6.01.03. http://www.code-aster.org/V2/doc/v9/fr/man_r/r6/r6.01.03.pdf. EDF (cit. on p. 24).

Bourgat, J-F., R. Glowinski, P. Le Tallec, and M. Vidrascu (1988). "Variational formulation and algorithm for trace operation in domain decomposition calculations". In: *Rapports de Recherche INRIA* 804 (cit. on p. 34).

Brandt, A. (1977). "Multi-level adaptive solutions to boundary-value problems". In: *Mathematics of Computation* 31.138, pp. 333–390 (cit. on pp. 2, 7).

Brandt, A., J. Brannick, K. Kahl, and I. Livshits (2011). "Bootstrap AMG". In: *SIAM Journal on Scientific Computing* 33.2, pp. 612–632 (cit. on p. 104).

Brezina, M., R. Falgout, S. MacLachlan, T. Manteuffel, S. McCormick, and J. Ruge (2004). "Adaptive Smoothed Aggregation (αSA)". In: *SIAM Journal on Scientific Computing* 25.6, pp. 1896–1920 (cit. on p. 104).

Cai, X-C., M. Dryja, and M. Sarkis (2003). "Restricted additive Schwarz preconditioners with harmonic overlap for symmetric positive definite linear systems". In: *SIAM Journal on Numerical Analysis* 41.4, pp. 1209–1231 (cit. on p. 22).

Cai, X-C. and D. Keyes (2002). "Nonlinearly preconditioned inexact Newton algorithms". In: *SIAM Journal on Scientific Computing* 24.1, pp. 183–200 (cit. on p. 74).

Cai, X-C. and M. Sarkis (1999). "Restricted additive Schwarz preconditioner for general sparse linear systems". In: *SIAM Journal on Scientific Computing* 21.2, pp. 792–797 (cit. on p. 22).

Campos, C., J. Román, E. Romero, and A. Tomás (2013). *SLEPc Users Manual*. Tech. rep. DSIC-II/24/02 - Revision 3.4. Universidad Politécnica de Valencia. URL: http://www.grycap.upv.es/slepc/ (cit. on p. 42).

Chabannes, V. (2013). "Vers la simulation des écoulements sanguins". PhD thesis. Université de Grenoble (cit. on p. 75).

Chartier, T., R. Falgout, V. Henson, J. Jones, T. Manteuffel, S. McCormick, J. Ruge, and P. Vassilevski (2003). "Spectral AMGe (ρAMGe)". In: *SIAM Journal on Scientific Computing* 25.1, pp. 1–26 (cit. on p. 34).

Chevalier, C. and F. Pellegrini (2008). "PT-Scotch: a tool for efficient parallel graph ordering". In: *Parallel Computing* 34.6, pp. 318–331. URL: http://www.labri.fr/perso/pelegrin/scotch/ (cit. on p. 42).

Chronopoulos, A. and C. Gear (1989). "s-step iterative methods for symmetric linear systems". In: *Journal of Computational and Applied Mathematics* 25.2, pp. 153–168 (cit. on p. 87).

Conen, L., V. Dolean, R. Krause, and F. Nataf (2014). "A coarse space for heterogeneous Helmholtz problems based on the Dirichlet-to-Neumann operator". In: *Journal of Computational and Applied Mathematics* 271, pp. 83–99 (cit. on p. 103).

CUDA Sparse Matrix library. NVIDIA. URL: https://developer.nvidia.com/cusparse (cit. on p. 39).

St-Cyr, A., M. Gander, and S. Thomas (2007). "Optimized multiplicative, additive, and restricted additive Schwarz preconditioning". In: *SIAM Journal on Scientific Computing* 29.6, pp. 2402–2425 (cit. on p. 103).

De Roeck, Y-H. (1993). "Nonlinear elasticity solved by a domain decomposition method on a hypercube". In: *Applied Numerical Mathematics* 12.5, pp. 459–471 (cit. on p. 74).

De Roeck, Y-H. and P. Le Tallec (1991). "Analysis and test of a local domain decomposition preconditioner". In: *Fourth International Symposium on Domain Decomposition Methods for Partial Differential Equations*. SIAM, pp. 112–128 (cit. on pp. 24, 34).

De Roeck, Y-H., P. Le Tallec, and M. Vidrascu (1992). "A domain-decomposed solver for nonlinear elasticity". In: *Computer Methods in Applied Mechanics and Engineering* 99.2, pp. 187–207 (cit. on p. 74).

De Sterck, H., R. Falgout, J. Nolting, and U. M. Yang (2008). "Distance-two interpolation for parallel algebraic multigrid". In: *Numerical Linear Algebra with Applications* 15.2-3, pp. 115–139 (cit. on p. 98).

De Sterck, H., U. M. Yang, and J. Heys (2006). "Reducing complexity in parallel algebraic multigrid preconditioners". In: *SIAM Journal on Matrix Analysis and Applications* 27.4, pp. 1019–1039 (cit. on p. 98).

De Sturler, E. and H. Van der Vorst (1995). "Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers". In: *Applied Numerical Mathematics* 18.4, pp. 441–459 (cit. on p. 87).

Deiterding, R. (2005). "Construction and application of an AMR algorithm for distributed memory computers". In: *Adaptive Mesh Refinement—Theory and Applications*. Springer, pp. 361–372 (cit. on p. 104).

Dohrmann, C. (2003). "A preconditioner for substructuring based on constrained energy minimization". In: *SIAM Journal on Scientific Computing* 25.1, pp. 246–258 (cit. on p. 31).

Dolean, V., P. Jolivet, and F. Nataf (2014). *An introduction to domain decomposition methods: algorithms, theory and parallel implementation*. SIAM, in preparation (cit. on pp. 13, 23, 35).

Dolean, V., P. Jolivet, F. Nataf, N. Spillane, and H. Xiang (2014). "Two-Level Domain Decomposition Methods for Highly Heterogeneous Darcy Equations. Connections with Multiscale Methods". In: *Oil Gas Sci. Technol. – Rev. IFP Energies nouvelles* 69.4, pp. 731–752.

Dolean, V., F. Nataf, R. Scheichl, and N. Spillane (2012). "Analysis of a two-level Schwarz method with coarse spaces based on local Dirichlet-to-Neumann maps". In: *Computational Methods in Applied Mathematics* 12.4, pp. 391–414 (cit. on p. 34).

Duff, I., A. Erisman, and J. Reid (1986). *Direct Methods for Sparse Matrices*. New York, NY, USA: Oxford University Press, Inc. (cit. on pp. 2, 7).

Duff, I., M. Heroux, and R. Pozo (2002). "An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum". In: *ACM Transactions on Mathematical Software* 28.2, pp. 239–267 (cit. on p. 39).

Engineering and Scientific Subroutine Library. IBM. URL: http://www-03.ibm.com/systems/power/software/essl/ (cit. on p. 86).

Falgout, R. and J. Schroder (2014). "Non-Galerkin Coarse Grids for Algebraic Multigrid". In: *SIAM Journal on Scientific Computing* 36.3, pp. C309–C334 (cit. on pp. 96, 104).

Falgout, R. and U. M. Yang (2002). "hypre: A library of high-performance preconditioners". In: *Computational Science—ICCS 2002*. Springer, pp. 632–641. URL: http://acts.nersc.gov/hypre/ (cit. on p. 42).

Farhat, C., M. Lesoinne, P. Le Tallec, K. Pierson, and D. Rixen (2001). "FETI-DP: a dual–primal unified FETI method—part I: A faster alternative to the two-level FETI method". In: *International Journal for Numerical Methods in Engineering* 50.7, pp. 1523–1544 (cit. on p. 33).

Farhat, C., M. Lesoinne, and K. Pierson (2000). "A scalable dual-primal domain decomposition method". In: *Numerical Linear Algebra with Applications* 7.7-8, pp. 687–714 (cit. on p. 33).

Farhat, C., J. Mandel, and F-X. Roux (1994). "Optimal convergence properties of the FETI domain decomposition method". In: *Computer Methods in Applied Mechanics and Engineering* 115.3, pp. 365–385 (cit. on p. 33).

Farhat, C. and F-X. Roux (1991). "A method of finite element tearing and interconnecting and its parallel solution algorithm". In: *International Journal for Numerical Methods in Engineering* 32.6, pp. 1205–1227 (cit. on pp. 24, 31).

— (1994). *Implicit Parallel Processing in Structural Mechanics*. Vol. 2. Computational Mechanics Advances 1. North-Holland (cit. on p. 30).

Fourier, J. (1822). *Théorie analytique de la chaleur*. Firmin Didot (cit. on p. 15).

Gabriel, E., G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. (2004). "Open MPI: Goals, concept, and design of a next generation MPI implementation". In: *Recent Advances in Parallel Virtual Machine and*

Message Passing Interface. Springer, pp. 97–104. URL: http://www.open-mpi.org/ (cit. on p. 86).

Gee, M., C. Siefert, J. Hu, R. Tuminaro, and M. Sala (2006). *ML 5.0 Smoothed Aggregation User's Guide*. Tech. rep. SAND2006-2649. Sandia National Laboratories. URL: http://trilinos.sandia.gov/packages/ml/ (cit. on p. 43).

Geuzaine, C. and J-F. Remacle (2009). "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities". In: *International Journal for Numerical Methods in Engineering* 79.11, pp. 1309–1331. URL: http://geuz.org/gmsh/ (cit. on pp. 61, 75).

Ghysels, P., T. Ashby, K. Meerbergen, and W. Vanroose (2013). "Hiding global communication latency in the GMRES algorithm on massively parallel machines". In: *SIAM Journal on Scientific Computing* 1.35, pp. 48–71 (cit. on p. 87).

Ghysels, P. and W. Vanroose (2013). "Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm". In: *Parallel Computing* (cit. on p. 89).

Giraud, L. and A. Haidar (2009). "Parallel algebraic hybrid solvers for large 3D convection-diffusion problems". In: *Numerical Algorithms* 51.2, pp. 151–177 (cit. on pp. 2, 7).

Gosselet, P. and C. Rey (2006). "Non-overlapping domain decomposition methods in structural mechanics". In: *Archives of Computational Methods in Engineering* 13.4, pp. 515–572 (cit. on p. 24).

Gosselet, P., C. Rey, and J. Pebrel (2013). "Total and selective reuse of Krylov subspaces for the resolution of sequences of nonlinear structural problems". In: *International Journal for Numerical Methods in Engineering* 94.1, pp. 60–83 (cit. on p. 104).

Gould, N., J. Scott, and Y. Hu (2007). "A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations". In: *ACM Transactions on Mathematical Software* 33.2 (cit. on p. 40).

Grigori, L., R. Stompor, and M. Szydlarski (2012). "A parallel two-level preconditioner for Cosmic Microwave Background map-making". In: *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*. SC12. IEEE Computer Society (cit. on p. 82).

Grote, M. and T. Huckle (1997). "Parallel preconditioning with sparse approximate inverses". In: *SIAM Journal on Scientific Computing* 18.3, pp. 838–853 (cit. on pp. 2, 7).

Gupta, A. (2000). *WSMP: Watson sparse matrix package—part II: Direct solution of general systems*. Tech. rep. 21888. IBM T.J. Watson Research Center (cit. on p. 40).

Gupta, A. and H. Avron (2000). *WSMP: Watson sparse matrix package—part I: Direct solution of symmetric systems*. Tech. rep. 21886. IBM T.J. Watson Research Center. URL: http://researcher.watson.ibm.com/researcher/view_group.php?id=1426 (cit. on p. 40).

Hecht, F. (2012). "New development in FreeFem++". In: *Journal of Numerical Mathematics* 20.3-4, pp. 251–266 (cit. on p. 60).

Hecht, F., S. Auliac, O. Pironneau, J. Morice, A. Le Hyaric, and K. Ohtsuka. *FreeFem++*. URL: http://www.freefem.org/ff++/ (cit. on pp. 60, 92).

Hénon, P., P. Ramet, and J. Roman (2002). "PaStiX: a high-performance parallel direct solver for sparse symmetric positive definite systems". In: *Parallel Computing* 28.2, pp. 301–321. URL: http://pastix.gforge.inria.fr/ (cit. on p. 40).

Henson, V. and U. M. Yang (2002). "*BoomerAMG*: A parallel algebraic multigrid solver and preconditioner". In: *Applied Numerical Mathematics* 41.1, pp. 155–177 (cit. on p. 82).

Heroux, M., R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, et al. (2005). "An overview of the Trilinos project". In: *ACM Transactions on Mathematical Software* 31.3, pp. 397–423 (cit. on p. 43).

Hestenes, M. and E. Stiefel (1952). "Methods of Conjugate Gradients for Solving Linear Systems". In: *Journal of Research of the National Bureau of Standards* 49.6, pp. 409–436 (cit. on p. 51).

Hoefler, T., P. Gottschling, A. Lumsdaine, and W. Rehm (2007). "Optimizing a conjugate gradient solver with non-blocking collective operations". In: *Parallel Computing* 33.9, pp. 624–633 (cit. on p. 87).

Hoefler, T., A. Lumsdaine, and W. Rehm (2007). "Implementation and performance analysis of non-blocking collective operations for MPI". In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. SC07. IEEE Computer Society (cit. on p. 87).

Jolivet, P., V. Dolean, F. Hecht, F. Nataf, C. Prud'homme, and N. Spillane (2012). "High-performance domain decomposition methods on massively parallel architectures with FreeFem++". In: *Journal of Numerical Mathematics* 20.4, pp. 287–302 (cit. on pp. 23, 34, 59, 110).

Jolivet, P., F. Hecht, F. Nataf, and C. Prud'homme (2013). "Scalable Domain Decomposition Preconditioners For Heterogeneous Elliptic Problems". In: *Proceedings of the 2013 ACM/IEEE conference on Supercomputing*. SC13. Best paper finalist. ACM, 80:1–80:11 (cit. on pp. 81, 82, 91, 110).

— (2014a). "Overlapping Domain Decomposition Methods with FreeFem++". In: *Domain Decomposition Methods in Science and Engineering XXI*. Vol. 98. Lecture Notes in Computational Science and Engineering. Springer, pp. 315–322 (cit. on p. 46).

— (2014b). "Scalable domain decomposition preconditioners for heterogeneous elliptic problems". In: *Scientific Programming* 22.2, pp. 157–171 (cit. on pp. 82, 110).

Karypis, G. and V. Kumar (1998). "A fast and high quality multilevel scheme for partitioning irregular graphs". In: *SIAM Journal on Scientific Computing* 20.1, pp. 359–392. URL: http://glaros.dtc.umn.edu/gkhome/views/metis (cit. on p. 42).

Kirk, B., J. Peterson, R. Stogner, and G. Carey (2006). "libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations". In: *Engineering with Computers* 22.3-4, pp. 237–254. URL: http://libmesh.sourceforge.net/ (cit. on p. 42).

Klawonn, A., M. Lanser, and O. Rheinbach (2014). "Nonlinear FETI-DP and BDDC Methods". In: *SIAM Journal on Scientific Computing* 36.2, A737–A765 (cit. on p. 74).

Klawonn, A. and O. Rheinbach (2007). "Inexact FETI-DP methods". In: *International Journal for Numerical Methods in Engineering* 69.2, pp. 284–307 (cit. on p. 33).

— (2010). "Highly scalable parallel domain decomposition methods with an application to biomechanics". In: *ZAMM - Zeitschrift für Angewandte Mathematik und Mechanik* 90.1, pp. 5–32 (cit. on pp. 3, 8, 82).

— (2012). "Deflation, projector preconditioning, and balancing in iterative substructuring methods: connections and new results". In: *SIAM Journal on Scientific Computing* 34.1, pp. 459–484 (cit. on p. 24).

Klawonn, A., O. Rheinbach, and O. Widlund (2008). "An analysis of a FETI-DP algorithm on irregular subdomains in the plane". In: *SIAM Journal on Numerical Analysis* 46.5, pp. 2484–2504 (cit. on p. 101).

Klawonn, A. and O. Widlund (2001). "FETI and Neumann-Neumann iterative substructuring methods: connections and new results". In: *Communications on Pure and Applied Mathematics* 54.1, pp. 57–90 (cit. on p. 33).

Kozubek, T., V. Vondrák, M. Menšík, D. Horák, Z. Dostál, V. Hapla, P. Kabelíková, and M. Čermák (2013). "Total FETI domain decomposition method and its massively parallel implementation". In: *Advances in Engineering Software* 60, pp. 14–22 (cit. on p. 82).

Lanczos, C. (1950). "An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators". In: *Journal of Research of the National Bureau of Standards* 45.4, pp. 255–282 (cit. on p. 93).

Le Tallec, P. (1994). "Domain decomposition methods in computational mechanics". In: *Computational Mechanics Advances* 1.2, pp. 121–220 (cit. on p. 22).

Lehoucq, R., D. Sorensen, and C. Yang (1998). *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. Vol. 6. Society for Industrial and Applied Mathematics. URL: http://www.caam.rice.edu/software/ARPACK/ (cit. on p. 41).

Leiserson, C. (1985). "Fat-trees: universal networks for hardware-efficient supercomputing". In: *IEEE Transactions on Computers* 100.10, pp. 892–901 (cit. on p. 92).

Li, X. (2005). "An Overview of SuperLU: Algorithms, Implementation, and User Interface". In: *ACM Transactions on Mathematical Software* 31.3, pp. 302–325. URL: http://crd-legacy.lbl.gov/~xiaoye/SuperLU/ (cit. on p. 42).

Li, X. and J. Demmel (2003). "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems". In: *ACM Transactions on Mathematical Software* 29.2, pp. 110–140 (cit. on p. 99).

Lions, P-L. (1988). "On the Schwarz alternating method. I". In: *First International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pp. 1–42 (cit. on p. 15).

Logg, A., K-A. Mardal, and G. Wells (2012). *Automated solution of differential equations by the finite element method*. Vol. 84. Springer, pp. 1–736 (cit. on p. 42).

Magoulès, F., F-X. Roux, and L. Series (2006). "Algebraic approximation of Dirichlet-to-Neumann maps for the equations of linear elasticity". In: *Computer Methods in Applied Mechanics and Engineering* 195.29, pp. 3742–3759 (cit. on p. 34).

Mandel, J. (1993). "Balancing domain decomposition". In: *Communications in Numerical Methods in Engineering* 9.3, pp. 233–241 (cit. on pp. 24, 29).

Mandel, J. and M. Brezina (1996). "Balancing domain decomposition for problems with large jumps in coefficients". In: *Mathematics of Computation of the American Mathematical Society* 65.216, pp. 1387–1401 (cit. on p. 31).

Mandel, J. and C. Dohrmann (2003). "Convergence of a balancing domain decomposition by constraints and energy minimization". In: *Numerical Linear Algebra with Applications* 10.7, pp. 639–659 (cit. on p. 31).

Math Kernel Library. Intel. URL: https://software.intel.com/en-us/intel-mkl (cit. on pp. 39, 92).

Mathew, T. (2008). *Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations*. Vol. 61. Lecture Notes in Computational Science and Engineering. Springer (cit. on p. 23).

Mooney, M (1940). "A Theory of Large Elastic Deformation". In: *Journal of Applied Physics* 11.9, pp. 582–592 (cit. on p. 70).

Nataf, F., F. Rogier, and E. de Sturler (1994). "Optimal interface conditions for domain decomposition methods". In: *Rapports Internes Centre de Mathématiques Appliquées de l'École Polytechnique* 301 (cit. on p. 34).

Nataf, F., H. Xiang, V. Dolean, and N. Spillane (2011). "A coarse space construction based on local Dirichlet to Neumann maps". In: *SIAM Journal on Scientific Computing* 33.4, pp. 1623–1642 (cit. on p. 34).

Newmark, N. (1959). "A method of computation for structural dynamics". In: *Journal of the Engineering Mechanics Division* 85.7, pp. 67–94 (cit. on p. 74).

Nicolaides, R. (1987). "Deflation of conjugate gradients with applications to boundary value problems". In: *SIAM Journal on Numerical Analysis* 24.2, pp. 355–365 (cit. on pp. 23, 56).

Nourtier-Mazauric, E. and E. Blayo (2010). "Towards efficient interface conditions for a Schwarz domain decomposition algorithm for an advection equation with biharmonic diffusion". In: *Applied Numerical Mathematics* 60.1, pp. 83–93 (cit. on p. 103).

Oorsprong, M., F. Berberich, V. Teodor, T. Downes, S. Erotokritou, S. Requena, E. Hogan, M. Peters, S. Wong, A. Gerber, E. Emeriau, R. Guichard, G. Yepes, K. Ruud, et al., eds. (2014). *PRACE Annual Report 2013*. Insight Publishers (cit. on p. 92).

Parks, M., E. de Sturler, G. Mackey, D. Johnson, and S. Maiti (2006). "Recycling Krylov subspaces for sequences of linear systems". In: *SIAM Journal on Scientific Computing* 28.5, pp. 1651–1674 (cit. on p. 104).

Parlett, B. (1998). *The Symmetric Eigenvalue Problem*. Vol. 20. Society for Industrial Mathematics (cit. on p. 41).

Pechstein, C. (2012). *Finite and Boundary Element Tearing and Interconnecting Solvers for Multiscale Problems*. Vol. 90. Lecture Notes in Computational Science and Engineering. Springer (cit. on p. 24).

Prud'homme, C. (2006). "A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations". In: *Scientific Programming* 14.2, pp. 81–110. URL: http://fenicsproject.org/ (cit. on p. 7).

Prud'homme, C., V. Chabannes, V. Doyeux, M. Ismail, A. Samake, and G. Pena (2012). "Feel++: A Computational Framework for Galerkin Methods and Advanced Numerical Methods". In: *ESAIM: Proceedings*. Vol. 38, pp. 429–455. URL: http://www.feelpp.org/ (cit. on pp. 42, 75).

Quarteroni, A. and A. Valli (1999). *Domain decomposition methods for partial differential equations*. Vol. 10. Clarendon Press (cit. on p. 23).

Rheinbach, O. (2009). "Parallel iterative substructuring in structural mechanics". In: *Archives of Computational Methods in Engineering* 16.4, pp. 425–463 (cit. on p. 24).

Rivlin, R. (1948). "Large elastic deformations of isotropic materials. IV. Further developments of the general theory". In: *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* 241.835, pp. 379–397 (cit. on p. 70).

Rixen, D. and C. Farhat (1997). "Preconditioning the FETI method for problems with intra- and inter-subdomain coefficient jumps". In: *Ninth International Conference on Domain Decomposition Methods*, pp. 472–479 (cit. on p. 30).

Roux, F-X. and C. Farhat (1998). "Parallel implementation of direct solution strategies for the coarse grid solvers in 2-level FETI method". In: *Contemporary Mathematics* 218, pp. 158–173 (cit. on p. 81).

Saad, Y. (1994). "ILUT: A dual threshold incomplete LU factorization". In: *Numerical Linear Algebra with Applications* 1.4, pp. 387–402 (cit. on pp. 2, 7).

— (2003). *Iterative Methods for Sparse Linear Systems*. SIAM (cit. on pp. 2, 7).

Saad, Y. and M. Schultz (1986). "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems". In: *SIAM Journal on Scientific and Statistical Computing* 7.3, pp. 856–869 (cit. on p. 52).

Samake, A. (2014). "Méthodes de décomposition de domaine pour les problèmes multi-échelles sur architectures hybrides". PhD thesis. Université de Grenoble (cit. on p. 101).

Sarkis, M. (2003). "Partition of Unity Coarse Spaces: Enhanced Versions, Discontinuous Coefficients and Applications to Elasticity". In: *Fourteenth International Conference on Domain Decomposition Methods*, pp. 149–158 (cit. on p. 23).

Schenk, O. and K. Gärtner (2004). "Solving unsymmetric sparse systems of linear equations with PARDISO". In: *Future Generation Computer Systems* 20.3, pp. 475–487 (cit. on p. 40).

— (2006). “On fast factorization pivoting methods for sparse symmetric indefinite systems”. In: *Electronic Transactions on Numerical Analysis* 23, pp. 158–179 (cit. on p. 40).
Schenk, O., K. Gärtner, and W. Fichtner (2000). “Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors”. In: *BIT Numerical Mathematics* 40.1, pp. 158–176. URL: http://www.pardiso-project.org/ (cit. on p. 40).
Schwarz, H. (1870). “Über einen Grenzübergang durch alternierendes Verfahren”. In: *Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich* 15, pp. 272–286 (cit. on p. 14).
Si, H. (2013). *TetGen: A Quality Tetrahedral Mesh Generator and 3D Delaunay Triangulator*. Tech. rep. 13. URL: http://wias-berlin.de/software/tetgen/ (cit. on p. 61).
Šístek, J., J. Mandel, B. Sousedík, and P. Burda (2013). “Parallel implementation of multilevel BDDC”. In: *Numerical Mathematics and Advanced Applications 2011*. Springer, pp. 681–689. URL: http://users.math.cas.cz/~sistek/software/bddcml.html (cit. on p. 104).
Smith, B., P. Bjørstad, and W. Gropp (2004). *Domain decomposition: parallel multilevel methods for elliptic partial differential equations*. Cambridge University Press (cit. on pp. 2, 7, 23).
Snir, M., S. Otto, D. Walker, J. Dongarra, and S. Huss-Lederman (1995). *MPI: The complete reference*. The MIT Press (cit. on pp. 1, 6, 38).
Spillane, N., V. Dolean, P. Hauret, F. Nataf, C. Pechstein, and R. Scheichl (2013). “Abstract robust coarse spaces for systems of PDEs via generalized eigenproblems in the overlaps”. In: *Numerische Mathematik* 126.4, pp. 741–700 (cit. on pp. 4, 9, 34).
Spillane, N. and D. Rixen (2013). “Automatic spectral coarse spaces for robust FETI and BDD algorithms”. In: *International Journal for Numerical Methods in Engineering* 95.11, pp. 953–990 (cit. on pp. 4, 9, 34, 35).
Sundar, H., G. Biros, C. Burstedde, J. Rudi, O. Ghattas, and G. Stadler (2012). “Parallel Geometric-Algebraic Multigrid on Unstructured Forests of Octrees”. In: *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*. SC12. IEEE Computer Society (cit. on pp. 3, 8).
Tang, J., R. Nabben, C. Vuik, and Y. Erlangga (2009). “Comparison of two-level preconditioners derived from deflation, domain decomposition and multigrid methods”. In: *Journal of Scientific Computing* 39.3, pp. 340–370 (cit. on p. 23).
Toselli, A. and O. Widlund (2005). *Domain decomposition methods: algorithms and theory*. Vol. 34. Series in Computational Mathematics. Springer (cit. on p. 23).
Vassilevski, P. (2008). *Multilevel block factorization preconditioners: matrix-based analysis and algorithms for solving finite element equations*. Vol. 10. Springer (cit. on p. 22).
Vassilevski, P. and U. M. Yang (2014). “Reducing communication in algebraic multigrid using additive variants”. In: *Numerical Linear Algebra with Applications* 21.2, pp. 275–296 (cit. on p. 87).
Wulf, W. and S. McKee (1995). “Hitting the memory wall: implications of the obvious”. In: *ACM SIGARCH Computer Architecture News* 23.1, pp. 20–24 (cit. on pp. 2, 8).

Pierre Jolivet

Méthodes de décomposition de domaine. Application au calcul haute performance.

Abstract

This thesis introduces a unified framework for various domain decomposition methods: those with overlap, so-called Schwarz methods, and those based on Schur complements, so-called substructuring methods. It is then possible to switch with a high-level of abstraction between methods and to build different preconditioners to accelerate the iterative solution of large sparse linear systems. Such systems are frequently encountered in industrial or scientific problems after discretization of continuous models. Even though these preconditioners naturally exhibit good parallelism properties on distributed architectures, they can prove inadequate numerical performance for complex decompositions or multiscale physics. This lack of robustness may be alleviated by concurrently solving sparse or dense local generalized eigenvalue problems, thus identifying modes that hinder the convergence of the underlying iterative methods a priori. Using these modes, it is then possible to define projection operators based on what is usually referred to as a coarse solver. These auxiliary tools tend to solve the aforementioned issues, but typically decrease the parallel efficiency of the preconditioners. In this dissertation, it is shown in three points that the newly developed construction is efficient: 1) by performing large-scale numerical experiments on Curie—a European supercomputer, and by comparing it with state of the art 2) multigrid and 3) direct solvers.

Keywords: linear algebra, preconditioner, high-performance computing, domain decomposition.

Résumé

Cette thèse présente une vision unifiée de plusieurs méthodes de décomposition de domaine : celles avec recouvrement, dites de Schwarz, et celles basées sur des compléments de Schur, dites de sous-structuration. Il est ainsi possible de changer de méthodes de manière abstraite et de construire différents préconditionneurs pour accélérer la résolution de grands systèmes linéaires creux par des méthodes itératives. On rencontre régulièrement ce type de systèmes dans des problèmes industriels ou scientifiques après discrétisation de modèles continus. Bien que de tels préconditionneurs exposent naturellement de bonnes propriétés de parallélisme sur les architectures distribuées, ils peuvent s'avérer être peu performants numériquement pour des décompositions complexes ou des problèmes physiques multi-échelles. On peut pallier ces défauts de robustesse en calculant de façon concurrente des problèmes locaux creux ou denses aux valeurs propres généralisées. D'aucuns peuvent alors identifier des modes qui perturbent la convergence des méthodes itératives sous-jacentes a priori. En utilisant ces modes, il est alors possible de définir des opérateurs de projection qui utilisent un problème dit grossier. L'utilisation de ces outils auxiliaires règle généralement les problèmes sus-cités, mais tend à diminuer les performances algorithmiques des préconditionneurs. Dans ce manuscrit, on montre en trois points que la nouvelle construction développée est performante : 1) grâce à des essais numériques à très grande échelle sur Curie—un supercalculateur européen, puis en le comparant à des solveurs de pointe 2) multi-grilles et 3) directs.

Mots-clefs : algèbre linéaire, préconditionneur, calcul haute performance, décomposition de domaine.